REAL-TIME IMPLEMENTATION AND EVALUATION OF AN ADAPTIVE SILENCE
DELETION ALGORITHM FOR SPEECH COMPRESSION

By

Christopher Rose

B. A. Sc., Simon Fraser University, 1987

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
DEPARTMENT OF ELECTRICAL ENGINEERING

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

August 1991

© Christopher Rose, 1991

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Electrical Engineering
The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date:

August 30, 1991

Abstract

The removal of nonessential acoustic material or silence from speech can significantly reduce the average bit rate required for speech coding. Silence deletion can be used to reduce the channel capacity required for speech transmission, or to reduce the storage requirements for voice mail applications.

This thesis describes the real-time implementation of a silence compression algorithm using a TMS320C25 digital signal processor. Substantial modifications were made to an earlier silence detection algorithm to enable its implementation in real time, as well as to improve the performance of the algorithm in a variety of speech environments. Up to 50% compression on single speaker speech is possible with our silence compression algorithm without significant loss of playback signal quality.

The silence compression algorithm was implemented in a way that enabled source coding techniques to be used to further compress the speech portions of the acoustic signal. The use of 4-bit ADPCM to code the speech resulted in an average bit rate below 19.2 kbit/s.

Table of Contents

Abstract	ii
List of Tables	vi
List of Figures	vii
Acknowledgement	viii
1 Introduction	1
1.1 Silence Compression	1
1.2 Review of Previous Work	3
1.3 Organization of Thesis	6
2 Silence Detection Algorithm Description and Development	8
2.1 Detection of Silence Intervals	8
2.1.1 Decision Criteria	8
2.1.2 Hysteresis in Energy Threshold	9
2.1.3 Classification of Speech/Silence Intervals	10
2.2 Adaptation to Noise and Signal Levels	12
2.3 Noise Insertion During Silence Intervals	16
2.4 Final Algorithm	17
2.5 Parameter Selection	19
2.5.1 Frame Size	19
2.5.2 Energy Thresholds	21
2.5.3 Zero Crossing Rate	21

2.5.4	Minimum Speech/Silence Intervals	23
2.5.5	Speech and Silence Interval Averages	23
2.5.6	Final Parameter Values	24
3	Real-time Implementation of Silence Compression Algorithm	26
3.1	Coding of Silence Intervals	27
3.1.1	Silence Code	27
3.1.2	Interval Length Coding	29
3.1.3	Interval Energy Coding	30
3.2	Coding of Speech Intervals	31
3.3	Long Term Signal Averaging	32
3.4	Data Transfer	34
3.5	Noise Playback	36
4	Algorithm Performance Evaluation	38
4.1	Speech Compression	38
4.2	Algorithm Complexity	39
4.3	Subjective Evaluation	39
5	Summary and Conclusions	44
5.1	Summary	44
5.2	Areas for Further Research	45
	Bibliography	47
A	Hardware Design	51
A.1	Hardware Description	51
A.2	Firmware Listings	60
B	Software Listings	65

B.1	Simulation Listings	65
B.2	Real-time Implementation Listings	72
B.2.1	Tms320c25 Program Listings	72
B.2.2	PC Host Program Listings	102

List of Tables

2.1	Optimum Algorithm Parameters for Various Framesizes	25
4.1	Compression Results for Various Frame Sizes	38
4.2	Algorithm Parameters for Processing Quiet Speech Sample	41
4.3	Algorithm Parameters for Processing Noisy Speech Sample	41
4.4	Listening Test Results Quiet Original Speech Sample	42
4.5	Listening Test Results Noisy Original Speech Sample	42

List of Figures

1.1	Speech Utterance Showing Silence Intervals	2
2.1	Effect of Speech Detection with Constant E-Thresholds	12
2.2	Effect of Speech Detection with Background Noise Level Adaptation	14
2.3	Effect of Speech Detection with Speech Level Adaptation	15
2.4	Effect of Speech Detection with Both Speech and Noise Level Adaptation	16
2.5	Block Diagram of the Silence Detection Algorithm	18
2.6	Percentage Silence for Various Framesizes with Fixed Thresholds	20
2.7	Percent Silence Deletion with Dual Thresholds	22
3.1	Format for Silence Interval Coding	27
3.2	Data Memory Pointer Organization	35
A.1	Block Diagram of Speech Compression System	51

Acknowledgement

I would like to thank my supervisor, Dr. Robert Donaldson, who suggested this topic, was a constant source of encouragement and helpful advice, and who provided financial support through NSERC operating grant A-3308.

I would also like to thank the many conscientious participants who assisted in the subjective evaluation of the speech compression algorithm.

Finally, I would like to acknowledge the financial assistance provided by the Natural Sciences and Engineering Research Council of Canada (NSERC), by the British Columbia Advanced Systems Institute (BC ASI), and by the University of British Columbia in the form of post-graduate scholarships.

Chapter 1

Introduction

1.1 Silence Compression

For many years, there has been active research to devise more bandwidth efficient methods of communicating and recording speech signals. With the evolution of digital technology, many methods of speech compression have been developed, generally utilizing one or more redundancies present in a speech signal [1,2]. Recent work has been done to find methods of coding toll quality speech at low bit rates (below 16 kb/s), as well as improving the speech quality at existing bit rates (32 and 64 kb/s) [3]. Modern coding techniques, such as vector quantization, can provide high compression rates (toll quality speech at 16 kb/s) but have a large computational requirement [4].

A simple and fairly effective method of speech compression involves the detection and removal of the numerous silence or non-speech intervals present during a conversation. These acoustically non-essential intervals exist in speech between sentences, phrases, and words; very short silence intervals are also present within words, such as before stop consonants. Studies have shown that for a normal two-way conversation, silence intervals comprise up to 65% of the utilization of each one-way communication channel [5,6].

Figure 1.1 shows a plot of the average short-term signal level for the speech utterance “A Simple Test”. The plot clearly shows the low level silence intervals before and after the phrase as well as between the words ‘simple’ and ‘test’. Careful observation also reveals the much shorter intra-word silence intervals just before the ‘p’ in ‘simple’ and before the final ‘t’ in ‘test’.

There are several applications for a system that can detect the silence periods within speech,

Figure 1.1: Speech Utterance Showing Silence Intervals

or conversely, the speech portions of an acoustic sample, as follows:

- The detection and removal of silence periods can greatly increase the storage capacity of speech recording and voice mail systems [7,8,9];
- Communication transmission systems can benefit from the detection and removal of silence periods; system transmission capacity can be increased by either not transmitting the silences as in a packet speech system [10], or by switching the channel to an active conversation or to a data channel during silence intervals [11];
- Speech recognition systems require that the start and end points of words be accurately isolated to facilitate the recognition process [12]; and
- Systems which perform adaptive filtering require periods of speech activity to be detected; applications include echo control systems for long distance communications, and adaptive

noise filtering systems.

For all of these applications, the utility of a speech detection system is strongly dependent on its complexity, and on the associated processing power that is required for its implementation.

An important consideration which can limit the range of silence detection or deletion applications is processing delay. A speech detector that is used for two-way communications applications requires a low processing delay and must inherently be implemented in real-time to reduce the end-to-end transmission delay. In a voice mail system, the real-time constraint is less critical; it would be feasible to detect the silence intervals off-line after initial recording. However, in most applications, there are definite advantages to real-time processing; excessive speech storage requirements are avoided, and the requirement for additional resources to reprocess the data off-line is reduced.

1.2 Review of Previous Work

Silence intervals in telephone conversations were first studied by Norwine and Murphy in 1938 [13]. In more recent years, research has been performed to develop accurate statistical models of silence intervals in conversational speech, as well as to determine the speech activity — talkspurt frequency trade-offs involved in developing a speech compression system [6,14].

In 1960, silence intervals were first utilized for increasing channel capacity in voice communications [5]. This system, known as *Time Assignment Speech Interpolation* (TASI), deallocated communication channels during periods of inactivity or silence for use with other conversations. Using this time division multiplexing method, TASI could carry more conversations than there were actual channels, thereby providing an apparent increase in channel capacity. The original TASI systems were implemented using analog electronics to perform the speech detection and circuit switching. Later systems took advantage of digital signal processing to combine the time division multiplexing with digital speech encoding techniques to achieve *Digital Speech Interpolation* (DSI) [15,16]. Combining new digital speech coding techniques with DSI provided a further increase in channel capacity. Most notably, much work has been done to combine

adaptive differential PCM (ADPCM) with DSI to obtain a 4:1 increase in channel capacity over that of conventional μ -law coded PCM [17,18].

The performance characteristics desired in a silence or speech detector varies greatly depending on the particular application at hand. A speech interpolation system typically has a large overhead associated with each silence interval to facilitate channel switching; it is therefore desirable to detect only the larger silence intervals to lower the talkspurt frequency and prevent excessive switching. Conversely, a speech storage facility typically has a very low silence coding overhead; therefore, the detection of very short intervals can increase the overall compression efficiency of the system. The majority of work done on silence or speech detectors has been tailored towards use in speech interpolation systems. However, much has been learned about silence detection from the TASI systems such as the effects of speech clipping, and the speech characteristics pertinent to speech detection.

The first speech detectors for TASI were based on an analog averaging detector operating on a 500 to 3000 Hz bandwidth, with a fixed threshold to differentiate speech and silence intervals [19]. To prevent the loss of weak syllables, a hangover was incorporated to prevent the release of the speech detector until the speech power remained below the threshold for 240 ms. This type of speech detector has two major drawbacks: the averaging speech detector is slow to respond to the onset of speech utterances, causing excessive speech clipping; and the hangover is inefficient, activating the speech detector for a portion of the acoustic signal which may or may not contain speech to compensate for the inability of the averaging detector to recognize low level speech.

Improved speech detectors were developed that used peak detection, rather than an averaging RMS detector [20,21]. The peak to average ratio of speech is 6 to 8 dB larger than the peak to average ratio of gaussian noise; therefore, if large peaks were detected, speech was assumed to be present. Since the speech was not averaged over a fixed period of time, the resulting detector is able to respond much more quickly to speech onsets. Fariello combined this peak detection approach with a zero crossing rate detector to enable the detection of low amplitude,

high frequency fricatives [20]. The peak detection scheme reported by Jankowski incorporated threshold adaptation based on the background noise level to improve the detection performance under varying conditions [21]. One draw back of this technique is that a hangover period of 150 to 160 ms is required to maintain activation of the speech detector between successive peaks in the speech signal.

Drago [22] developed a speech detector based on a speech model of a high frequency carrier modulated by a low frequency envelope. Speech — non-speech decisions were based on the dynamic variations of the low frequency envelope or energy of the acoustic signal, rather than the absolute level as with conventional speech detectors. The resulting detector had better performance for speech onsets, and was less sensitive to acoustic signal level than conventional detectors. A hangover period of 250 ms was used.

Un and Lee [23] developed a speech — silence discriminator based on bit alternation rates of delta modulation recorded speech. The high sampling rates used in delta modulation (typically 24,000 to 32,000 samples per second) enable silence intervals to be detected by a high rate of 0/1 bit alternations which typically result when the delta modulator dithers to track a very low level signal such as which occur between speech utterances. Cho and Un [24] later used this speech detector to reduce the overall bit rate in delta modulation coded speech.

De Souza [12] developed an adaptive, self-normalizing silence detector for use in continuous speech recognition applications. The detector was based on the concept of "pattern recognition", performing a statistical test on a series of parameters calculated for each frame of acoustic signal. The parameters included frame energy, number of zero-crossings, and autocorrelation calculations. The silence detector required a 500 ms training interval of background noise at the beginning of a speech recording in order to function correctly. However, the parameters for typical silence frames were updated continuously in order to adapt to any variations in the background noise.

Recent work has been performed towards the development of a simple yet effective method of detecting and utilizing silence intervals for speech compression [25,26]. Compression rates

of 35 to 40% were obtained using a relatively noisy recording of a single talker reading from a prepared text. In this previous work, the author developed an adaptive method for adjusting the speech – silence energy thresholds based on the level of the background noise. However, using the adaptive threshold scheme proposed, the algorithm performance for relatively noise-free speech is inconsistent; there is a tendency for the thresholds to approach zero, resulting in reduced compression. Since compression of good quality speech is desirable, the algorithm as described [25] is problematic.

1.3 Organization of Thesis

The primary objective of this thesis is to describe the design, real-time implementation, and evaluation of a silence compression algorithm. Also described is the substantial work done to enhance the robustness, playback quality, and efficiency of the basic silence detection algorithm, which serves as a starting point for this thesis research. Our evaluations are based on subjective speech quality assessments using listening tests, measured compression statistics, and implementation simplicity.

Chapter 2 details the development of the silence interval detection algorithm, which is based in part on one developed earlier [25,26]. Described are the substantial steps taken to ensure efficient compression, good quality speech, and robust performance in various operating environments, as well as means to enable real-time implementation using standard electronic components. The chapter includes a section on the selection of the optimum algorithm parameters.

Chapter 3 details the real-time implementation of our silence compression algorithm. Included are discussions on the coding of silence intervals into the speech data, source coding of the speech data to further improve compression, and the manipulation of the acoustic data to facilitate a real-time speech storage or transmission system.

The evaluation of the silence compression algorithm is detailed in Chapter 4. The results of subjective evaluations of the speech compression system are presented, along with an analysis

of compression efficiency and implementation complexity. The results show that up to 50% of single speaker continuous speech can be deleted from 32 kbit/s ADPCM speech with minimal loss in quality.

Chapter 5 summarizes the contributions of the thesis, and suggests some topics for further research.

Chapter 2

Silence Detection Algorithm Description and Development

This chapter presents the algorithm used to detect the silence periods within the speech samples. The basic algorithm was adapted from one developed previously [25]; however, substantial modifications were devised to accomplish the following:

1. make the algorithm easier to implement in a real-time system;
2. improve the compression performance and playback quality; and
3. improve the robustness of the algorithm for use with less noisy speech.

2.1 Detection of Silence Intervals

In processing the input speech, the samples are divided into non-overlapping intervals or *frames* of size K samples. Each frame will be classified as either speech or silence based on decision criteria calculated on the data within that frame, and on the classification of previous frames.

2.1.1 Decision Criteria

Two basic decision criteria are used to classify the input frames as either speech or silence: short-time average amplitude and mean zero crossing rate. The short-time average amplitude criteria is designed to act on the large variation of signal levels present in the speech waveform [2]. It will detect the large levels during vowels and vowel-like consonants. Zero crossing rate facilitates detection of the low level, high frequency components of speech such as unvoiced fricative and stop consonants. Gan found [25] zero crossing rate to be an effective means of detecting these portions of speech, particularly with noisy speech signals. A zero crossing is

defined as a position within the signal where the sign of the signal level changes between two successive samples.

For each frame of speech F_i , the average signal amplitude E_i and the number of zero crossings Z_i for the interval are calculated as follows:

$$E_i = \frac{1}{K} \sum_{j=1}^K |X_{Ki+j}| \quad (2.1)$$

$$Z_i = \sum_{j=1}^K S(X_{Ki+j}) \quad (2.2)$$

where

$$S(X_n) = \begin{cases} 0 & \text{if } \text{sgn}(X_n) = \text{sgn}(X_{n-1}) \\ 1 & \text{otherwise.} \end{cases}$$

The function $S(X_n)$ has a value of 1 only if X_n and X_{n-1} have different signs, which occurs if and only if a zero crossing falls between them. The value of Z_i , as calculated by equation 2.2, is equal to the total number of zero crossings that occur within the frame F_i . The notation X_{Ki+j} represents the j th digital sample within the i th frame of size K .

The short-time amplitude average is used as a decision criteria, instead of the short-time energy (in which the square of each sample is summed) to reduce the sensitivity of varying input signal levels due to the squaring operation [27], and to simplify the calculation of the short-time energy. In this thesis, the average amplitude E_i calculated from equation 2.1 shall be loosely referred to as the signal energy.

Initially, each frame is evaluated as being either speech or silence based solely on that frame's average energy and number of zero crossings. In particular, the frame F_i is classified as speech if $E_i \geq \text{E-Threshold}$, or if $Z_i \geq \text{Z-Threshold}$. The frame is classified as silence if neither of these two conditions hold true.

2.1.2 Hysteresis in Energy Threshold

The basic algorithm described above can be improved by taking advantage of some of the speech waveform features. The start of speech utterances display a larger average signal energy than

the termination. Hence, the implementation of a dual threshold scheme can be used to increase the threshold for speech onsets to prevent triggering by noise, and to decrease the threshold during speech to prevent clipping which occurs when the threshold is too large. The hysteresis nature of this dual threshold scheme also prevents frequent speech/silence transitions when the speech level is near the energy threshold. Such transitions increase overhead for silence interval coding.

Hysteresis is easily implemented through the use of two energy thresholds. Once a state of silence has been reached, the larger threshold is used in identifying silence-to-speech transitions. Similarly, once a state of speech is present, the smaller threshold is used in identifying speech-to-silence transitions.

There are times, however, when it is advantageous to maintain the smaller threshold for a short period after silence has been detected. A low threshold facilitates detection of weak stop consonants, which are always preceded by a short period of silence. This effect is increasingly important as the framesize K increases, because the consonant energy becomes blurred in the surrounding silence. As the framesize becomes smaller, the speech energy becomes more responsive to short-term variations, and this hang-over is not necessary. In fact, due to the larger number of silence intervals present when computing with the smaller framesizes, incorporating a significant hang-over could maintain the energy threshold at its low value, effectively cancelling the energy threshold hysteresis and reducing the effectiveness of our silence compression algorithm.

2.1.3 Classification of Speech/Silence Intervals

Operation of the algorithm in the basic form, as described in the previous section, can give very good performance results as measured by both compression and speech quality. However, as the frame size gets smaller, the silence intervals have a tendency to occur in rapid succession. This effect can be harmful if the overhead involved with coding silence intervals is high. The overhead can be reduced somewhat if the absolute speech/silence decisions are based on previous frames

as well as on the current frame.

Previous work has considered the use of both a minimum speech constraint [25] and a minimum silence duration [28]. The minimum speech constraint is designed to classify as silence those input samples initially classified as speech, but which are too short to convey useful information. The minimum silence constraint is designed to classify as speech the silence intervals too short to be efficiently handled by the speech compression application. By using both of these criteria, the speech compression algorithm can be tailored to suit various application needs.

Care must be exercised in the selection of the minimum speech and minimum silence criteria. Increasing the minimum silence duration alone will not degrade the reconstructed speech quality, but can drastically reduce the efficiency of the compression algorithm. Increasing the minimum speech duration, on the other hand, will have the reverse effect; the compression capabilities of the algorithm will increase at the cost of causing the potential for increased speech clipping.

Implementation of minimum speech and minimum silence constraints is relatively straightforward using the system concept of speech and silence states. In the silence state, any additional silence frames simply add to the duration of the current silence interval. There must be at least N consecutive speech frames to transfer to the speech state, where N is the minimum speech constraint. If there are fewer than N consecutive speech frames, then those frames are discarded, the current silence interval duration is increased, and the silence state remains in effect. Likewise, if the speech state is active, speech frames are directly added to the outgoing data stream. There must be at least M consecutive silence frames to begin another silence interval and transfer to a silence state, where M is the minimum silence interval duration. If the number of consecutive silence intervals is less than M , the silence frames are added to the outgoing data stream, and the speech state remains in effect.

2.2 Adaptation to Noise and Signal Levels

To this point, we have treated the value for E-Threshold as being constant. In a speech detection system, it is very difficult to obtain optimum performance with a constant value for E-threshold. Speech levels, as well as background noise levels, can vary from call to call. Even within the duration of a single call, the background noise level can vary significantly.

In a fixed threshold system, a greater percentage of the speech will fall below threshold and be treated as silence as the input speech level decreases. Similarly, a louder speech level will cause a lower percentage of silence detection. Figure 2.1 shows the effect of using a constant E-threshold with varying input speech levels, as a function of framesize. The quieter -6 dB

Figure 2.1: Effect of Speech Detection with Constant E-Thresholds

speech sample has a much higher % silence deletion than the $+6$ dB sample.

Any use of constant values for E-threshold must compromise between different speech and background noise levels. Such compromise leads to excessive speech clipping in some cases,

and highly non-optimum speech detection performance in others. In order to achieve robust performance, the energy threshold value should be adaptive to both the speech signal and background noise levels.

Silence detection performance when the energy thresholds are made to adapt to the background noise level has been investigated in previous work [25], where the energy threshold was calculated as follows:

$$E\text{-Threshold}_i = K_T \cdot E\text{-Avg}_i. \quad (2.3)$$

In (2.3), $E\text{-Avg}_i$ is the current long term average background noise energy level, and K_T is a threshold constant. A preset threshold value K_T is multiplied by the current average background noise energy $E\text{-Avg}_i$ to produce the actual threshold. We found that this approach works fairly well, particularly with noisy speech; however, as the background noise level drops to zero, so do the threshold values. This results in a much lower percentage of silence detection than would be expected for the given input sample. Figure 2.2 shows the percentage silence detection for 3 input speech samples, each with a different level of background noise. The algorithm works well for moderate amounts of background noise, but shows reduced compression for the sample with no added noise.

A first approach to making the background noise level adaptation work with low levels of background noise is to place a fixed lower limit on the energy threshold. However, such a fixed lower limit would be sub-optimal in an environment where speech levels can fluctuate widely. A better solution is to make this lower limit dependent on the input speech level, or in other words, make the energy threshold adapt to both the average speech level as well as to the average background noise level.

To make the energy thresholds adaptive to both the speech and background noise levels, two calculations for the current energy threshold are made. One is based solely on the background noise level, while the other is based only on the current averaged speech level. The actual threshold used is then calculated as

$$E\text{-Threshold} = K_T \cdot \text{MAX}(\text{SilAvg}, \text{SpeechAvg}/13) \quad (2.4)$$

Figure 2.2: Effect of Speech Detection with Background Noise Level Adaptation

where

$$\text{MAX}(X, Y) = X \text{ if } X \geq Y; \text{ else } Y. \quad (2.5)$$

In (2.4), $SilAvg$ is the current long term average background noise energy level, $SpeechAvg$ is the current long term average speech energy level, and K_T is a threshold constant. If the input speech sample is very noisy, the background noise level threshold calculation will dominate, and the algorithm will behave in a similar manner to that of one reported earlier [25]. However, if the speech sample is relatively noise-free, the speech level threshold calculation will dominate, and will prevent the threshold levels from approaching zero. The effect, then, is to place a lower limit on the energy threshold that is dependent on the input speech level. This approach enables good compression statistics on loud or high level recordings to be maintained without causing excessive clipping on quiet or low level recordings.

Figure 2.3 shows the effectiveness of speech level adaptation with varying levels of input

Figure 2.3: Effect of Speech Detection with Speech Level Adaptation

speech. As can be seen, there is less than a 0.5% difference in silence detection when the input speech is varied over a 12dB dynamic range. This is a dramatic improvement over the case of no adaptation as shown in figure 2.1, where the three graphs vary by up to 23% in silence detection.

Figure 2.4 shows the combined effect of adaptation based on both speech and noise levels for speech samples with varying amounts of background noise. At a high background noise level, the graph in figure 2.4 is similar to that in figure 2.2, indicating that the noise level dominates in the dual adaptive system under this condition. For the speech sample with very little background noise, the dual adaptive system performs well and prevents the thresholds from dropping to zero, as is the case with noise-only adaptation. The middle graph corresponds to a voice sample with an amount of noise which approximates the maximum specified for a long distance telephone link. This background noise level was chosen as the cross-over point between

Figure 2.4: Effect of Speech Detection with Both Speech and Noise Level Adaptation

noise level and speech level adaptation.

2.3 Noise Insertion During Silence Intervals

In tests with silence compression, some of the silence intervals were quite noticeable during playback, because of the replacement of the deleted acoustic material with complete silence. Gan [25] determined that this effect could be reduced by replacing the pure silence during playback with a copy of the original background noise.

In a practical situation, a copy of the original background noise may not be available, or at least it may not be practical to utilize extra resources for its transmission or storage. In this thesis, the possibility of inserting random noise during playback was investigated.

During playback, randomly generated noise was inserted during the silence intervals at a level equal to the background noise at the time of the recording. The background noise level

is obtained as part of the compression algorithm for use in determining the appropriate energy threshold. This same level is, therefore, available for coding in the outgoing data stream for use in generating the noise in the playback process. This approach allows the playback noise level to adapt to the background noise level at the time of recording, throughout the speech segment.

White noise was used during playback due to its simplicity in generation, and its generally acceptable sound quality during playback. The algorithm could easily be adapted to incorporate some type of digital filter to colour the noise if such filtering would improve the perceived playback quality.

2.4 Final Algorithm

Figure 2.5 shows a block diagram of the complete algorithm that was implemented in both a real-time system, and in an off-line software simulator for evaluating the effect of the various parameters.

The modified silence detection algorithm has a number of parameters that can be used to tailor its performance for different operating environments. The algorithm parameters include:

1. Frame Size
2. Upper and Lower Energy Thresholds, and Threshold Hangover
3. Zero Crossing Threshold, and
4. Minimum Speech and Silence Interval Duration.

Also of importance is the interval over which long term averages of the speech and silence energies are calculated. The effect of each of these parameters on algorithm performance is presented in the next section.

Figure 2.5: Block Diagram of the Silence Detection Algorithm

2.5 Parameter Selection

Optimization of the algorithm parameters was performed using a PC-based ‘simulation’ program on pre-recorded files. This approach enabled accurate evaluation of the parameter effects using identical input speech files. Also, the speech files could be modified in a controlled manner to simulate various input speech samples. Once parameters were characterized and selected, the results of the compression trials could then be compared to those obtained using a real-time speech compression system.

Informal listening tests were performed by the author to characterize the effects of the parameters on the silence detection algorithm performance. With the effect of each parameter known, the task of selecting a parameter set to provide a pre-determined level of performance for a particular application is simplified. It was found, however, that there are strong inter-dependencies between the various parameters, and that the selection of one influenced the selection of the others. The parameters can be adjusted to give very good performance under different conditions (ie, noise levels) although a well chosen set will give acceptable although non-optimum (in some cases) performance all-round.

This section describes the results of the tests to evaluate parameter values, and presents the effects on speech quality and compression of each of the algorithm parameters. Considerations in selecting optimum parameter values are also presented.

2.5.1 Frame Size

The number of speech samples used to calculate the short term statistics of the input signal is an important consideration in the performance of the silence detection algorithm. As the framesize becomes smaller, the short term energy is averaged over a smaller number of samples, and hence has a much larger and more rapid variation than if averaged over a larger number of samples. This larger variation gives a much greater ability to detect low level consonants which might otherwise be ‘blurred’ into the surrounding noise, and clipped. As the framesize becomes larger, small silence intervals are averaged into the surrounding speech (and vice versa) and as

a result, the talkspurt frequency decreases. This decreased talkspurt frequency, however, comes at a cost of decreased silence detection, and possible increased speech clipping.

The dependance of short term energy variance on framesize implies that the most appropriate energy threshold values are also dependent on framesize. Figure 2.6 shows a plot of detected silence percentage vs framesize for different energy thresholds. The graphs were compiled by running the algorithm on a prepared speech sample, with an added background noise energy level of 20 units. The algorithm was run in basic form, with no minimum speech, minimum silence, or zero-crossing criteria, and no hysteresis. The actual energy thresholds used to generate this graph were calculated by multiplying the threshold values displayed in figure 2.6 by 20, the average energy of the background noise level in the input speech sample. The graph clearly

Figure 2.6: Percentage Silence for Various Framesizes with Fixed Thresholds

shows that for a given energy threshold, a larger silence percentage can usually be detected by using a smaller framesize. The one exception is the plot for Threshold = 1.5, where for small

framesizes the large interframe variation in silence interval energy causes an increased portion of the silence intervals to exceed the threshold level, and be classified as speech.

An even more substantial advantage to using a smaller framesize can be seen if the reconstructed speech quality is taken into account. The line labeled *Maximum* in figure 2.6 shows the maximum percent silence deletion that can be obtained for each framesize before noticeable speech clipping occurs. As the frame size gets larger, more speech clipping is apparent for a given threshold multiplier; therefore, smaller frames can use larger thresholds and hence yield more compression for a given speech playback quality.

2.5.2 Energy Thresholds

Figure 2.6 shows that the optimum energy threshold for the silence detection algorithm is strongly dependent on the framesize used to calculate the short term statistics. Figure 2.7 shows a similar graph, except with the data obtained using a dual threshold algorithm with hysteresis. This graph shows a similar dependence of the optimum thresholds on the framesize. For each of the thresholds shown, the ratio of Min to Max is the same. The maximum percent silence deletion line is very similar to that seen in figure 2.6, being the same for larger framesizes, and only 1 to 2 percent higher at a framesize of 4. Figures 2.6 and 2.7 show that adding hysteresis provides very little enhancement to the maximum silence deletion possible by the algorithm. However, the simulations with hysteresis had a significant reduction in the number of silence intervals obtained (or number of talkspurts). Such reduction is important in a practical system where there is a finite overhead involved in coding each silence interval, or where there is a need to keep the talkspurt frequency low.

2.5.3 Zero Crossing Rate

For large framesizes, the zero-crossing rate criterion is required to avoid classifying the low amplitude consonants as silence, resulting in clipped speech. This fact was reported earlier [25].

Figure 2.7: Percent Silence Deletion with Dual Thresholds

For small framesizes, the use of the zero-crossing rate criterion can actually be counter-productive. A small framesize implies only a few zero-crossings are possible within each frame, and the large variability inherent in small sample populations causes an excessive portion of the background noise to be classified as speech. Small framesizes, however, show a much larger variation in the short-term energy than a large framesize for the same speech waveform. As such, the weak consonants are more pronounced from the background noise, removing the need for the zero-crossing criterion.

When the zero-crossing rate criterion is required, the selection of the zero-crossing threshold is a compromise; too low a value results in an excessive portion of silence being classified as speech, while too large a value results in the threshold rarely being exceeded, reducing its effectiveness significantly. It was found that a threshold corresponding to slightly less than half of the successive samples in a frame exhibiting zero-crossings provided the best compromise.

2.5.4 Minimum Speech/Silence Intervals

The inclusion of minimum speech and silence interval durations enables the tailoring of the silence detection algorithm to a particular application. Situations requiring a low talkspurt frequency can be facilitated by using a larger minimum silence duration. In this way, shorter silence durations coalesce, resulting in fewer talkspurts.

There is a definite dependence between the two minimum interval duration parameters. A large minimum silence duration can lead to a very low percentage silence unless the minimum speech duration is also increased. If the minimum speech duration remains small, short bursts of acoustic signal will be detected as speech, and a full silence interval duration is required before silence classification continues. Likewise, if the minimum speech duration is made long, sections of speech will not be detected if interspersed with short silence intervals, leading to excessive speech clipping.

Tests show that there is an improvement in reconstructed speech quality if a smaller frame-size is used with larger minimum speech/silence durations than if larger framesizes are used with smaller minimum intervals. However, the best silence percentage and reconstructed speech quality results are obtained when using a small framesize with small minimum speech and silence durations. The disadvantage of this choice of parameters is an increased talkspurt rate, making it unsuitable for some applications such as speech interpolation or other applications where there is a relatively large overhead associated with each silence interval.

2.5.5 Speech and Silence Interval Averages

The silence detection algorithm relies on long term speech and background noise energy averages to adapt the energy thresholds for optimum performance. The appropriate number of acoustic samples required to form these averages is an important consideration. The number of samples must be large enough to 'average out' any short term variations in either the speech or silence portions of the acoustic signal, but small enough to enable tracking of the long term variations.

The silence interval energy varies quite slowly, and hence only a short sampling of the silence

is needed to give a fairly accurate long term silence energy. Tests show that good silence interval energy tracking can be obtained by averaging 128 ms of silence. The speech signal, on the other hand, is quite varied in amplitude, and changes rapidly. This implies that a relatively long sampling of speech is required to give an accurate value of average energy. Testing revealed that up to one second of speech information is required.

Although the time period required to evaluate the long-term averages does not change with framesize, the number of frames required does change. This effect creates a disadvantage with using too small a framesize, as the number of frames (and hence the overhead) required to calculate the long-term statistics becomes excessive.

Silence interval samples near a silence-to-speech transition typically have a relatively high short-term average amplitude. To prevent these samples from corrupting the long-term silence interval average energy, a parameter ECRIT was used [25]. A short-term average (ie, a frame) was included in the long-term average only if the frame did not exceed $\text{ECRIT} \cdot \text{SilAvg}$, where SilAvg is the long-term silence energy average. The optimum value for ECRIT was determined by adjusting this parameter, and comparing the compression results with those obtained with a fixed energy threshold. The speech sample used had a constant level of background noise added to a voice sample recorded with a fairly constant long-term speech average. It was found that the best ECRIT value varied with the framesize K , from 1.20 for $K=128$ to 1.60 for $K=4$.

2.5.6 Final Parameter Values

To determine an optimum set of parameters for a speech storage system, a speech file with constant levels of speech and background noise was used. The background noise level was set to correspond to the cross-over point from background noise to speech signal adaptation.

With this file, a dual energy threshold silence detector with fixed threshold levels was used to determine the optimum threshold levels for various framesizes. The threshold levels were increased until speech clipping was just apparent, as judged by careful listening by the author. The compression statistics for these optimum thresholds were noted for later reference.

The algorithm was then altered to one that used threshold adaptation based on background noise level only. The value of ECRIT could be varied until the compression statistics matched those of the fixed threshold system. The fixed case can be viewed as a limiting case where the background noise adaptation remains constant, due to a constant background noise level; any disturbances due to inclusion of speech onsets in the background noise average would be absent.

Finally, the algorithm was altered to include both speech signal and background noise level adaptation; the speech file was changed to one containing no background noise, but the same recorded speech level. This enabled the SpeechScale parameter to be tuned until compression performance matched that with using fixed amplitude thresholds. A value for SpeechScale of 13 produced the best results, with an initial speech amplitude average estimate of 250.

The required sample lengths for the long term averages for both the speech and background noise levels were determined by analyzing the energy tracking for various sample lengths. The long term statistics were averaged over an interval long enough to remain roughly constant for a constant level speech sample, but not so long as to disable tracking for a speech sample whose amplitude exhibited long-term variability. It was found that the speech should be averaged over approximately one second, and that background noise should be averaged over 128 ms.

The optimum algorithm parameters for various framesizes are shown in table 2.1, where SilAvgLen and SpcAvgLen are the number of frames used in the calculation of the long term silence and speech averages respectively.

Frame Size	Min	Max	ECrit	SilAvgLen	SpcAvgLen
128	2.10	2.52	1.20	8	64
64	2.40	2.88	1.20	16	128
32	2.60	3.12	1.23	32	256
16	3.10	3.72	1.30	64	512
8	3.40	4.08	1.42	128	1024
4	3.90	4.68	1.60	256	2048

Table 2.1: Optimum Algorithm Parameters for Various Framesizes

Chapter 3

Real-time Implementation of Silence Compression Algorithm

The silence detection algorithm presented in chapter 2 provides a means for distinguishing silence intervals from the acoustically essential speech periods. To implement this algorithm into a practical system, the following concerns need resolution:

1. Silence intervals must be efficiently coded to enable speech compression, while allowing accurate reproduction during playback;
2. Coding of speech intervals to achieve overall compression better than that obtainable using conventional 64 kbit/s PCM must be accommodated;
3. An efficient means for computing the long-term average amplitude of the silence and speech intervals must be developed;
4. Compressed speech data must be efficiently transferable to the desired storage or transmission medium; and
5. White noise of appropriate energy level must be generated in an efficient manner for insertion in the silence intervals during playback.

For this thesis, the silence compression algorithm was implemented in real time using a TMS320C25 digital signal processor. The compressed speech data was transferred to an IBM PC/AT compatible host computer for storage on a hard disk. This chapter describes the details of this real-time implementation. The details of the hardware to implement this system are contained in appendix A; the software listings for both the real-time system and the PC based simulator appear in appendix B.

3.1 Coding of Silence Intervals

The silence detection algorithm described in chapter 2 divides the input speech into non-overlapping speech and silence intervals. To achieve speech compression with this algorithm, the silence intervals must be efficiently coded into the output data stream for low bit rate storage or transmission. During playback, the silence interval codes are detected by the decoder, and are used to accurately reproduce the original silence interval.

The important parameters of a silence interval are its location within the speech segment, its length, and its average background noise amplitude. Figure 3.1 shows a silence code format

Silence Code	Interval Length	Interval Mean Amplitude
--------------	-----------------	-------------------------

Figure 3.1: Format for Silence Interval Coding

which contains the necessary parameters for accurate silence interval reproduction. The location of the silence intervals are indicated by the position of the silence codes within the speech data.

To simplify the implementation for this thesis, each of the three fields of the silence interval code was allocated a 16-bit word. As discussed below, the field lengths could easily be modified in different applications to provide more efficient coding.

3.1.1 Silence Code

The silence interval code begins with a unique bit sequence to distinguish it from the speech data. In previous research, a series of repetitive speech codes was used as the silence code [25]. By making the number of repetitions long enough, the probability of the silence code occurring naturally in the speech data was brought close to zero. While this method does provide a certain amount of flexibility, there are some drawbacks, including:

- the silence code has the possibility of becoming quite long, increasing the overhead in coding the silence intervals;
- the probability of a naturally occurring silence code cannot be made zero, even with an arbitrarily long code sequence; and
- the insertion of a repetitive code following a natural occurrence of a portion of the code will cause incorrect decoding (the addition of a non-removable dummy code, prior to the repetitive silence code, was required).

For this work, a single unlikely bit sequence was used as the silence code. In particular, the code was a concatenation of two consecutive speech samples that swing from the negative to positive limits of the speech coder. For example, with Mu-law coded PCM speech, the silence code would be 0080 hex; the most negative value followed by the most positive value. At 8000 samples per second, this represents a large amplitude 4kHz signal, which is virtually impossible over a 3.4kHz bandlimited channel. In all the tests and experiments performed, this silence code-word never occurred naturally.

To allow for the possibility of a silence code occurring naturally in the speech data, such an occurrence would be followed by a special marker sequence in a manner similar to that used in character oriented communication protocols such as IBM's BISYNC [29,30]. During playback, the decoder searches for the silence codes and examines the word immediately following for the marker. If it exists, the marker code is removed, and the preceding silence code is treated as speech data. If the silence code – marker code combination occurs naturally in the speech data, the encoder would insert another marker code after the silence code. In the decoder, the second marker code would be removed, leaving the correct speech data for playback. For this work, the marker sequence was an all zero word.

A second marker sequence was used to indicate the end of the speech data file. This marker allows the speech compression system detect the end of speech data, and terminate speech playback. In the real-time system, this reduces the amount of communication necessary

between the silence compression processor and the host file storage system. The end-of-speech-file marker was the hex word FFFF.

3.1.2 Interval Length Coding

The second portion of the silence interval code specifies the length of the silence interval. While there are a number of ways to specify the silence interval length, the simplest is to use the number of frames of silence that comprise the interval, with a limit on the maximum length of the interval to some large value. This was the method used in this thesis, and it provided acceptable results. There is a disadvantage to this method, however, particularly when using a small frame size; the number of bits required to code long intervals can be excessive.

There are a number of possible ways to improve the efficiency of the interval length coding, including:

- A codebook could be provided that contains a variety of pre-determined interval lengths. A short code that represents the length closest to the actual interval length would then be sent to the decoder.
- The interval lengths could be compressed in a manner similar to PCM companding. This would give high resolution for shorter intervals and still provide for larger intervals using relatively few bits to encode the length.
- The interval length field in the silence interval code could be made variable in size. A small number of bits could encode the more common short intervals, with a single bit indicating additional length fields to encode the less common larger intervals.

Although none of these methods were investigated in this thesis, their implementation would be relatively easy in the real-time environment.

As was mentioned previously, the interval length field may also contain special markers. In this thesis, two special markers were used (based on a 16-bit field width):

1. A code to indicate that the silence code had occurred naturally in the speech data (0000 hex); and
2. A code to mark the end of the speech data file (FFFF hex).

A zero length interval was used to mark a naturally occurring silence code as it was easy to verify, and had no meaning as a real interval length. To mark the end of the speech data, the largest possible interval length code was used; the natural occurrence of this code was prevented (eg, during a very long silence interval) by limiting the interval length to FFFE hex. Even with this limiting, it was possible to represent up to 32 seconds of contiguous silence with 8000 samples per second and 4 samples per frame.

3.1.3 Interval Energy Coding

The final field of the silence interval code provides information regarding the average background noise amplitude during the interval. Such knowledge enables a more accurate reproduction of the original silence interval during playback. This is an optional field, however, and is not required for all applications of the speech compression algorithm.

For this thesis, the average background noise level calculated during silence detection was inserted into the final field of the silence interval code. The use of this average prevented the local variations of playback noise which would occur if the energy of a single frame only were used; however, the average did enable tracking of long term variations.

The use of a 16 bit field to transmit the amplitude average was somewhat excessive, but did simplify implementation, since the TMS320 uses 16 bit words. The amplitude average for the silence interval could be directly coded using only 8 bits (allowing values of 0 to 255), and could most likely be reduced even further through the use of non-linear quantizing (giving higher resolution to the lower levels, and less resolution to the larger levels). Other alternatives include the following:

- A pre-determined level of noise could be inserted during playback, requiring no interval average amplitude code. With this method, the noise will not adapt to the source

background noise level; however, an acceptable system may still be possible for most applications.

- A reduced number of background noise level updates could be used, instead of being sent with every silence interval code. It may suffice to send an update at fixed intervals, or only when the noise level changes significantly.
- The background noise average amplitude could be encoded differentially. Since the background noise level typically remains fairly constant during a speech sample, fewer bits may be required if only the difference in noise levels was encoded for each interval.

All of these modifications could easily be implemented in a real-time system.

Through the use of a combination of the coding techniques mentioned for both the interval length and average amplitude, it would be possible to reduce the silence coding overhead by over 50%. This is of particular importance for smaller frame sizes, where the current coding overhead can be in excess of 5% of the original speech sample.

3.2 Coding of Speech Intervals

The previous section provides a means for compressing the silence portions of speech into silence codewords. Further overall compression can be obtained by source coding the speech portions of the input signal using a common form of coding such as PCM, companded PCM, ADPCM, and ADM [1,2]. The main concern in coding the speech intervals is in dealing with the silence intervals; coding schemes which depend on previous speech samples could lose this vital information during silence intervals, thereby preventing accurate signal reconstruction during playback. Previous work has been done to show that some sample-to-sample dependent coding methods are robust, and can regain tracking following silence periods [25]. For non-memory coding schemes, such as PCM, this problem becomes trivial. Additionally, the speech coding algorithm must work with the silence compression algorithm in a one-pass format for real-time implementation.

For this thesis, the input speech was in μ -law companded PCM format, a standard form for PCM networks which uses inexpensive speech CODECs. This μ -law PCM speech needs to be converted to linear PCM for the silence detection algorithm; however, in a basic system speech is stored in μ -law form.

For speech coding schemes which use intra-sample dependencies, the decoder parameters will not adapt during the silence intervals; the original speech samples are not present, and the inserted silence interval samples are generated in linear PCM form. If no compensating action is taken, the speech decoder will contain the parameters from the end of the most recent speech interval at the start of the new speech interval. Mistrack between the speech coder and the decoder will occur and may cause an audible distortion in the reconstructed speech.

One solution to this problem is to reset the coder parameters at the start of each new potential speech interval (if the speech frame is detected as a silence interval, then the coder will be reset for the start of the next frame). Resetting the speech coder places it into a known state, which can then be duplicated in the decoder at the start of each new speech interval. This resetting eliminates the mistracking between the coder and decoder.

This coder resetting method was implemented using 4-bit ADPCM speech coding [31,32] providing a noticeable improvement over a direct ADPCM implementation. Audible clicks that were present at the start of reconstructed speech intervals when resetting was not used, were removed.

3.3 Long Term Signal Averaging

The silence detection algorithm described in chapter 2 requires the calculation of long term signal averages in order to track varying speech and background noise levels. Since these averages are calculated over a large number of samples, special consideration must be given to the time required to calculate the average and the memory required to store the samples. This section describes the methods used to minimize both the calculation time and the memory storage requirement.

The equation for calculating the long-term average $A(t)$ at time t of N frame amplitude averages E_i is

$$A(t) = \frac{S(t)}{N} \quad (3.1)$$

where

$$S(t) = \sum_{i=t-N+1}^t E_i.$$

One sees from (3.1) that as N becomes large, the processing required to calculate $A(t)$ in this direct form becomes excessive for a real-time implementation.

Calculation of the long-term averages can be simplified by noting that

$$A(t+1) = \frac{1}{N}(E_{t+1} + S(t) - E_{t+1-N}). \quad (3.2)$$

Equation 3.2 requires only 2 additions, as opposed to the N additions that are required for (3.1). The use of an integer based calculation for $S(t)$ in (3.2) maintains full precision, thereby sustaining accuracy of the average $A(t)$. The computational requirements of (3.2) can be further reduced if the value N is a power of two. In this case, the required division is reduced to a simple arithmetic shift.

Independent of which method is used to actually calculate the long-term averages, it is required that all N samples contained in the average are temporarily stored in memory. This storage requirement places a large demand on memory, particularly if N is large, as is the case for calculating average speech amplitude. A block averaging scheme was used to reduce these memory requirements while still giving an accurate value for the long-term average speech energy. The block averaging scheme was not used in the calculation of the long-term silence average, as the number of frames used in this calculation was much less than for the speech average.

The block averaging scheme first averages a smaller number of frame energies into block averages, and then uses these block averages to calculate the overall average speech amplitude. For example, if there are n blocks, each containing m frame amplitude averages (where $m \cdot n =$

N), the block averages B_i are evaluated as follows:

$$\begin{aligned} B_1 &= \text{Avg}(E_1, \dots, E_m) \\ B_2 &= \text{Avg}(E_{m+1}, \dots, E_{2m}) \\ &\vdots \\ B_n &= \text{Avg}(E_{N-m+1}, \dots, E_N). \end{aligned}$$

The long-term speech average $A(t)$ is then calculated as $\text{Avg}(B_1, \dots, B_n)$. This average is updated at time $t + m$, using an equation similar to 3.2, when the block average B_{n+1} becomes available.

Since each frame amplitude average E_i is used only once in the calculation (to determine the block average), it is not required to be stored in memory. Only the block averages, which are used to update the long-term speech average $A(t)$, are required to be saved. Using this method, the memory requirements are reduced substantially.

The one disadvantage of this scheme is that the long-term speech average is updated only after m frames of speech are detected. However, if m is chosen appropriately, the effects of this do not significantly alter the performance of the silence detection algorithm. For this thesis, m was chosen so that the speech average was updated after every 32 ms of speech.

3.4 Data Transfer

One problem in implementing the speech compression algorithm in a real-time environment is developing a method to efficiently transfer the silence compressed speech data between the DSP system and the storage or transmission medium. This problem is further compounded in that speech data must be temporarily stored before the silence detection algorithm can determine whether or not such data includes silence. The amount of speech to be stored depends on the values of MinSil and MinSpeech, but will be at least one frame in duration. The data cannot be transferred to the host computer until the current speech/silence interval classification is completed.

For this thesis, the data memory used to store the speech data has two access ports, one for the DSP system, and one for the host computer. The DSP system can store the incoming speech data while simultaneously calculating the frame statistics. Depending upon the subsequent speech/silence decisions, this data can either be transferred to the host (speech interval), or over-written with new data (silence interval).

Figure 3.2 shows the pointers that were used to manage the dual port data memory. The

Figure 3.2: Data Memory Pointer Organization

Current Data Pointer (CDP) and New Data Pointer (NDP) are managed by the DSP system, and the SAVED pointer is managed by the host system. The NDP is communicated to the host system when it is updated, to notify the host that data has been classified and is ready for storage or transmission.

Incoming speech samples are stored at the CDP memory location while the short-term averages are calculated; the CDP is advanced one location after storage. If the samples are

later classified as speech, the NDP is advanced to the first location past the speech data; the host system then knows this data has been classified, and either stores the data or transmits it. If the acoustic data is classified as silence, the CDP is reset back to the NDP (the end of the last SPEECH interval) and the silence code is written; new speech data is then stored after the silence code. If adjacent intervals of silence are detected, the CDP is reset to before the previous speech code (NDP) and the speech code is over-written with the new code with the new silence interval length. This continues until a speech interval is detected, at which point the NDP is advanced.

3.5 Noise Playback

For optimum performance, the silence detection algorithm described in chapter 2 requires the insertion of random noise in the silence intervals during playback. For this thesis, white noise was generated with an average amplitude equal to that in the original silence interval.

White noise was generated by first generating uniform random numbers based on the linear congruential method of number generation, where each new random number X_{i+1} is calculated as

$$X_{i+1} = (aX_i + c) \bmod m \quad (3.3)$$

where a , c , and m are constants selected to give the desirable statistics to the random numbers [33]. Much work has been performed on the selection of these constants to provide randomness in all bits of X_i [34], however, for this work, ease of implementation and long cycle period are considered to be the most important characteristics. For this thesis, $a = 67081293$, $c = 14181771$, and $m = 67108864 = 2^{26}$. This set of constants have been shown to have a long cycle time as well as good randomness properties [35]. Having m as a power of 2 reduces the modulus operation to a bit masking operation, greatly simplifying the implementation of the random number generator in a real-time environment.

The generated uniform random variables are transformed into gaussian random variables using a method based on the central limit theorem. By summing 12 uniform [0,1] random

variables as

$$G_n = \sum_{i=1}^{12} X_i \quad (3.4)$$

the resulting random variable G_n approximates a gaussian random variable with a mean of 6 and a variance of unity [36]. By subtracting 6 from each number G_n generated, zero mean white noise is produced. Using this method, the white noise generator had a period of 16,777,216 or 2^{24} samples, which corresponds to 34.95 minutes at 8000 samples per second. Such time interval is more than sufficient to prevent the repetition from being audible during playback.

The level required for the playback noise is inserted as part of the silence interval code, allowing the playback noise to change to accommodate changes in the background noise level in the original speech sample. To create the actual playback noise sample, the generated random value G_n was multiplied by the background noise level amplitude average and a correction factor of 1.248. The correction factor compensates for the fact that the average amplitude is used in the noise level calculation, rather than the true average energy which corresponds to the random variable variance. The correction factor was determined experimentally.

Chapter 4

Algorithm Performance Evaluation

To evaluate the silence compression algorithm, the following factors need to be considered:

- the amount of speech compression obtainable;
- the complexity involved in implementing the algorithm in real-time; and
- the quality of the reconstructed speech to determine the degradation to the speech signal through compression.

This chapter evaluates the silence compression algorithm described in chapters 2 and 3 based on these factors.

4.1 Speech Compression

Table 4.1 presents the maximum compression obtainable using our silence compression algorithm on a recording of single speaker continuous speech from prepared text, using the algorithm parameters presented in table 2.1. Percent compression is defined as percent silence minus over-

Frame Size	% Silence	% Speech	% Overhead	% Compression
128	41.46	58.54	0.19	41.27
64	44.02	55.98	0.26	43.75
32	46.64	53.36	0.48	46.16
16	50.62	49.38	1.23	49.39
8	53.73	46.27	2.77	50.96
4	58.51	41.49	7.41	51.10

Table 4.1: Compression Results for Various Frame Sizes

head.

Table 4.1 shows that speech compression of over 50% is attainable using our algorithm. The actual compression value depends on the block size. These results compare well with those of earlier work, where speech compression of 35 to 40% was obtained using a frame size of 100 samples [25]. Table 4.1 shows that almost 10% more speech compression can be obtained by using a framesize of 4 samples over a framesize of 128 samples. Using advanced speech coding methods (such as ADPCM) with the silence compression algorithm, average bit rates less than 19.2 kb/s are obtainable.

4.2 Algorithm Complexity

The speech compression algorithm described in the previous two chapters provides a high level of speech compression for a relatively low cost in computing power. For this thesis, the silence compression algorithm was implemented in real-time using a TMS320c25 digital signal processor running at 40 MHz. With this system, only 25% of the processor capabilities was utilized for the algorithm. It is expected that a base version of the silence detection algorithm, which would exclude zero-crossing, MinSpeech, and MinSil parameters, and exhibit a reduced level of threshold adaptation, could possibly be implemented using a more general purpose and lower cost processor, or even with a custom integrated circuit. One advantage of using the higher power digital signal processor is the excess resources available for additional speech coding for higher compression rates, or for networking or other supervisory tasks that may be necessary for a particular speech compression application or system.

4.3 Subjective Evaluation

The subjective evaluation was performed through listening tests based on executions of the silence compression algorithm on the following speech sample:

It is claimed that young children up to the age of about seven or eight years are incapable of grasping the abstract fundamental that number and volume remain constant even through changes in the outward appearance of the object. For many

years the work of Piaget and his colleagues in Geneva has profoundly influenced the education of the young child.

This portion of text was taken from that used in earlier work [25], but was recorded for these tests by the author, whose native language is English. The author's accent is representative of English speaking western Canadians.

Two experiments were performed to evaluate the silence detection algorithm performance; one using a somewhat noisy speech sample, and one using a speech sample with very little background noise. The noisy speech sample was created from the low-noise original speech sample by adding a fixed level of random noise, roughly equivalent in level to that of a noisy long-distance telephone link. Speech samples based on both silence playback and noise playback during silence classified segments were included in both of the experiments.

The experimental speech files were generated in a way that enabled a meaningful comparison of the results. It was suspected, from informal listening tests by the author, that a larger percentage of speech could be removed using a smaller framer size, while maintaining a constant playback speech quality. As such, the experimental speech segments consisted of processed speech samples with frame sizes of 128, 32, and 8, with the MIN and MAX thresholds selected so there was a small amount of speech clipping in each case (the thresholds were slightly larger than the *optimal* values shown in table 2.1). Also included were processed speech samples with frame sizes of 32 and 8 with thresholds selected to give roughly the same amount of speech removal as did the slightly clipped sample with a frame size of 128.

Tables 4.2 and 4.3 give the parameters used to generate the experiment files for the low noise level and moderate noise level speech samples, respectively. Also included in these tables are the compression statistics obtained for each of the files. The % Overhead columns show the overhead in compressing the silence intervals, based on a 6-byte code per silence interval.

The files created using the parameters shown in tables 4.2 and 4.3 were combined with the corresponding original speech files into two experimental sessions. In each, two random speech samples, along with the originals, were repeated; once towards the start of each listening session,

File #	Frame Size	Min	Max	ECrit	% Silence	% Overhead	% Compression
1,2	128	2.50	3.00	1.20	44.56	0.21	44.35
3,4	32	2.20	2.64	1.23	44.50	0.44	44.07
5,6	8	2.00	2.40	1.42	44.80	2.20	42.60
7,8	32	2.90	3.48	1.23	48.57	0.55	48.07
9,10	8	3.50	4.20	1.42	56.06	3.00	53.07
11,12	8	3.20	3.84	1.42	53.81	2.80	51.01

Table 4.2: Algorithm Parameters for Processing Quiet Speech Sample

File #	Frame Size	Min	Max	ECrit	% Silence	% Overhead	% Compression
1,2	128	2.50	3.00	1.20	44.56	0.21	44.35
3,4	32	2.20	2.64	1.23	44.50	0.44	44.07
5,6	8	2.00	2.40	1.42	44.80	2.20	42.60
7,8	32	2.90	3.48	1.23	48.57	0.55	48.07
9,10	8	3.50	4.20	1.42	56.06	3.00	53.07

Table 4.3: Algorithm Parameters for Processing Noisy Speech Sample

and again towards the end. This repetition of samples enabled a check on the consistency of the listeners. In all, there were 16 test samples in each session. Each of the two sessions began with the original samples to orient the test participants.

The test participants were given the following instructions:

The purpose of this subjective listening test is to evaluate the overall effect of silence compression on speech.

Please state the degree of acceptability of the test samples as recorded messages on a scale of 1 to 5. 1 means that the sample is unacceptable, and 5 denotes the highest degree of acceptance. Rate each sample on its own, rather than by comparison with other samples.

The test consists of 16 samples, each of a 30 second duration. The samples are numbered 1 through 16, with the current sample number being displayed at the top of the computer screen.

For your orientation, the first sample is an unprocessed original.

The test samples are played through the telephone handset. Pressing any key on the keyboard will begin the playback of each speech sample.

Please record your evaluations on the response sheet provided.

Tables 4.4 and 4.5 summarize the results of the listening tests for the low noise level and moderate noise level speech samples, respectively. For the low noise speech samples, 15 test participants evaluated the playback speech quality (10 male, 5 female); for the noisy speech samples, there were 25 test participants (16 male, 9 female). All of the participants were adults.

File #	Frame Size	Mean Score		Standard Deviation	
		Silent	Noisy	Silent	Noisy
	Original	4.9/4.7		0.3/0.4	
1,2	128	3.8	3.6	1.1	1.1
3,4	32	4.0	4.0	0.9	0.8
5,6	8	4.1/4.0	4.3	0.6/0.9	0.6
7,8	32	3.1/3.6	3.7	0.9/1.1	0.8
9,10	8	3.3	3.1	0.8	1.0
11,12	8	3.7	3.2	0.9	1.0
Mean Standard Deviation				0.79	0.88

Table 4.4: Listening Test Results Quiet Original Speech Sample

File #	Frame Size	Mean Score		Standard Deviation	
		Silent	Noisy	Silent	Noisy
	Original	4.8/4.6	3.5/3.5	0.4/0.7	0.7/0.9
1,2	128	2.4	3.2	1.2	1.0
3,4	32	2.5	3.6	1.2	0.9
5,6	8	2.9/2.6	3.5	1.1/0.9	0.9
7,8	32	2.3/2.4	3.5	1.0/1.2	0.9
9,10	8	2.2	3.1	0.9	1.0
Mean Standard Deviation				0.96	0.90

Table 4.5: Listening Test Results Noisy Original Speech Sample

The experimental results shown in tables 4.4 and 4.5 show that the listening groups as a whole were fairly consistent; the repeated speech samples of file numbers 5 and 7, and the original differ by at most 0.5 on a scale of 1 to 5. There is a relatively high standard deviation of approximately one rating unit on almost all of the processed speech samples test scores. This

indicates that the tolerance to the degradation in speech quality varies considerably between individuals. It was also noticed that while some individuals were less tolerant of speech clips and speech—silence transitions, others appeared to object to higher levels of background noise.

For the low noise level test samples, the difference in scores between noise playback and silence playback was small. This was expected, as the playback noise level was set to equal the original noise level which was low in this case. The results in table 4.4 do show that 44% removal of acoustic material can be obtained from a high quality speech sample, while degrading the speech quality less than 0.8 rating units. Removal of approximately 50% acoustic material results in a degradation of 1.1 rating unit at 8 sample block size.

For the moderate noise level test samples, the addition of noise at a level roughly equal to the original background noise upon playback improved the perceived speech quality by approximately one rating unit. The overall quality rating of the processed speech samples with noise insertion on playback varied by at most 0.4 rating units from the noisy, unprocessed original, even when well over 50% of the acoustic signal was removed.

Chapter 5

Summary and Conclusions

5.1 Summary

This thesis described the development, real-time implementation, and evaluation of a new silence deletion algorithm for speech compression. The algorithm was implemented in real time using a TMS320C25 digital signal processor, and produced speech compression rates of over 50% in highly active samples of readings by a single speaker from prepared texts. The reconstructed speech quality was acceptable for both noisy and relatively noise free speech. Improved reconstruction quality was obtained by inserting noise during the silence intervals on playback; such improvement was particularly apparent for noisy speech. Using our real-time system with 4-bit ADPCM voice coding, it would be possible to transmit speech with an average transmission rate of below 19.2 kbit/s, with a relatively low computational effort. Encoding and playback processing delays would be one or two milliseconds, which is insignificant.

Modifications made to the silence detection algorithm reported by Gan [25] to improve real-time performance also enhanced the performance of our silence detection. The framesize for short-term statistics was reduced to enhance compression. The algorithm amplitude thresholds were made adaptive to the speech level as well as to the background noise level, to provide more robust and consistent performance for both noisy and noise free speech. A minimum silence duration parameter was added to enhance flexibility in tailoring the algorithm to particular applications. While some work was done towards selecting an optimum set of algorithm parameters, it was found that the parameters need to be optimized for a particular application and operating environment. It was also determined that one set of parameters can give relatively good performance with respect to speech compression and reconstructed speech quality under

a variety of operating conditions.

The primary contributions of this thesis research then, are as follows:

- a new algorithm was developed for a real-time speech storage and playback system utilizing compression of silence intervals;
- the algorithm was implemented on one circuit board, in real-time, using standard electronic hardware;
- optimization of a speech detection algorithm and associated parameters for robust performance under different operating conditions and high speech compression was achieved;
- evaluation of the silence compression algorithm was completed based on subjective listening tests, and compression statistics.

5.2 Areas for Further Research

This thesis was primarily concerned with the real-time implementation of a silence deletion algorithm. While some performance evaluation and parameter optimization was done, further work in these areas is required.

The silence detector performance in the presence of different types of background noise (different spectra and statistics), particularly burst noise common in telephone environments, needs to be investigated. As well, further testing of the compression performance, robustness, and playback quality for various speakers and speech environments is needed. In particular, our algorithm needs to be tested with different speakers, languages other than English, and a variety test material.

This thesis identified the characteristics of each of the parameters of the silence detection algorithm, and used these characteristics to determine a good set of parameters for the speech compression application that our experimental setup emulated. Further work is required, however, to develop a set of optimum algorithm parameters for use in various speech compression environments.

For this thesis, the silence interval code consisted of three 16-bit words to store the silence code, the silence interval length, and the silence interval average amplitude. Some suggestions for optimizing the silence interval code were presented in this thesis; these ideas and others should be investigated. Optimization of the silence interval code becomes more important as the frame size decreases and the number of silence intervals increases; under these circumstances, the silence interval coding overhead can account for up to a 7% reduction in the overall compression.

The silence compression algorithm was implemented using μ -law coded PCM, and 4-bit ADPCM coded speech. The investigation of the algorithm performance with other speech coding techniques should be performed.

Finally, the implementation of our silence compression algorithm with either a lower cost processor, or custom integrated circuits should be investigated.

Bibliography

- [1] J. L. Flanagan *et al.*, "Speech coding," *IEEE Transactions on Communications*, vol. COM-27, pp. 710-737, Apr. 1979.
- [2] T. W. Parsons, *Voice and Speech Processing*. McGraw-Hill Book Company, 1986.
- [3] N. S. Jayant, "High quality coding of telephone speech and wideband audio," *IEEE Communications Magazine*, pp. 10-20, Jan. 1990.
- [4] A. Gersho and V. Cupperman, "Vector quantization: a pattern-matching technique for speech coding," *IEEE Communications Magazine*, pp. 15-21, Dec. 1983.
- [5] K. Bullington and J. M. Fraser, "Engineering aspects of TASI," *The Bell System Technical Journal*, vol. 38, pp. 353-364, March 1959.
- [6] P. T. Brady, "A statistical analysis of on-off patterns in 16 conversations," *The Bell System Technical Journal*, vol. 47, pp. 73-91, Jan. 1968.
- [7] J. L. Flanagan, J. D. Johnston, and J. W. Upton, "Digital voice storage in a microprocessor," *IEEE Transactions on Communications*, vol. COM-30, pp. 336-345, Feb. 1982.
- [8] P. Mermelstein, "Voice message systems," *IEEE Communications Magazine*, pp. 8-10, Dec. 1983.
- [9] N. F. Maxemchuk, "An experimental speech storage and editing facility," *The Bell System Technical Journal*, vol. 59, pp. 1383-1395, Oct. 1980.
- [10] S. A. Webber, C. J. Harris, and J. L. Flanagan, "Use of variable-quality coding and time-interval modification in packet transmission of speech," *The Bell System Technical Journal*, vol. 56, pp. 1569-1573, Oct. 1977.

- [11] H. P. Stern, "Design issues relevant to developing an integrated voice/data mobile radio system," *IEEE Transactions on Vehicular Technology*, vol. 39, pp. 281–288, Nov. 1990.
- [12] P. de Souza, "A statistical approach to the design of an adaptive self-normalizing silence detector," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-31, pp. 678–684, June 1983.
- [13] A. C. Norwine and O. J. Murphy, "Characteristic time intervals in telephonic conversation," *The Bell System Technical Journal*, vol. 17, pp. 281–291, Apr. 1938.
- [14] H. H. Lee and C. K. Un, "A study of on-off characteristics of conversational speech," *IEEE Transactions on Communications*, vol. COM-34, pp. 630–637, June 1986.
- [15] S. J. Campanella, "Digital speech interpolation," *COMSAT Technical Review*, vol. 6, pp. 127–158, Spring 1976.
- [16] K. Feher *et al.*, *Telecommunications Measurements, Analysis, and Instrumentation*. New Jersey: Prentice-Hall Inc., 1987.
- [17] G. G. Langenbucher, "Efficient coding and speech interpolation: principles and performance characterization," *IEEE Transactions on Communications*, vol. COM-30, pp. 769–779, Apr. 1982.
- [18] J. Ishida *et al.*, "High-gain DSI equipment using VLSI signal processor," in *IEEE Globecom 84 Conference Proceedings*, (Atlanta, Georgia), pp. 292–297, Nov. 1984.
- [19] H. Miedema and M. G. Schachtman, "TASI quality — effect of speech detectors and interpolation," *The Bell System Technical Journal*, vol. 41, pp. 1455–1473, July 1962.
- [20] E. Fariello, "A novel digital speech detector for improving effective satellite capacity," *IEEE Transactions on Communications*, vol. COM-20, pp. 55–60, Feb. 1972.
- [21] J. A. Jankowski, Jr., "A new digital voice activated switch," *COMSAT Technical Review*, vol. 6, pp. 159–178, Spring 1976.

- [22] P. G. Drago, A. M. Molinari, and F. C. Vagliani, "Digital dynamic speech detectors," *IEEE Transactions on Communications*, vol. COM-26, pp. 140–145, Jan. 1978.
- [23] C. K. Un and H. H. Lee, "Voiced / unvoiced / silence discrimination of speech by delta modulation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-28, pp. 398–407, Aug. 1980.
- [24] D. H. Cho and C. K. Un, "Hybrid companding delta modulation with silence detection," in *IEEE Global Communications Conference Proceedings*, (Miami, Florida), pp. 1340–1344, Nov. 1982.
- [25] C. K. Gan, "Efficient speech storage via compression of silence periods," Dept. of Electrical Engineering MASC Thesis, University of British Columbia, Dec. 1984.
- [26] C. K. Gan and R. W. Donaldson, "Adaptive silence deletion for speech storage and voice mail applications," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-36, pp. 924–927, June 1988.
- [27] L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals*. New Jersey: Prentice-Hall Inc., 1978.
- [28] J. G. Gruber, "A comparison of measured and calculated speech temporal parameters relevant to speech activity detection," *IEEE Transactions on Communications*, vol. COM-30, pp. 728–738, Apr. 1982.
- [29] A. S. Tanenbaum, *Computer Networks*. New Jersey: Prentice-Hall Inc., 1981.
- [30] D. P. Bertsekas and R. G. Gallager, *Data Networks*. New Jersey: Prentice-Hall Inc., 1987.
- [31] W. R. Daumer *et al.*, "Overview of the ADPCM coding algorithm," in *IEEE Globecom 84 Conference Proceedings*, (Atlanta, Georgia), pp. 774–777, Nov. 1984.

- [32] J. Reimer, M. McMahan, and M. Arjmand, "32-kbit/s ADPCM with the TMS32010," in *Digital Signal Processing Applications with the TMS320 Family*, ch. 17, pp. 469–530, Texas Instruments Incorporated, 1986.
- [33] R. P. Chambers, "Random-number generation on digital computers," *IEEE Spectrum*, vol. 4, pp. 48–56, Feb. 1967.
- [34] S. K. Park and K. W. Miller, "Random number generators: good ones are hard to find," *Communications of the ACM*, vol. 31, pp. 1192–1201, Oct. 1988.
- [35] E. F. Casas, "OFDM/FM for mobile radio data communication," Dept. of Electrical Engineering PhD Thesis, University of British Columbia, Sep. 1989.
- [36] M. Schwartz and L. Shaw, *Signal Processing: Discrete Spectral Analysis, Detection, and Estimation*. McGraw-Hill Book Company, 1975.
- [37] *Second Generation TMS320 User's Guide*. Texas Instruments Incorporated, 1987.
- [38] L. Pagnucco and C. Erskine, "Companding routines for the TMS32010/TMS32020," in *Digital Signal Processing Applications with the TMS320 Family*, ch. 5, pp. 169–212, Texas Instruments Incorporated, 1986.

Appendix A

Hardware Design

A.1 Hardware Description

Figure A.1 shows a block diagram of the hardware used to implement the silence compression algorithm. The system uses a TMS320C25 digital signal processor [37] to perform the actual

Figure A.1: Block Diagram of Speech Compression System

silence compression, and an IBM-PC/AT compatible computer to facilitate the storage of the processed data. The majority of the silence compression algorithm is contained in the DSP system, with the PC acting as a host to store, manipulate, and retrieve the compressed speech data.

There are five basic blocks that form the interface between the TMS320 and the PC. They are as follows:

1. A control register to control the operation of the TMS320 processor;

2. A status register to monitor the status of the TMS320 processor;
3. A bi-directional communications port for passing critical data between the TMS320 and the PC in a synchronized format;
4. Dual-port RAM for passing general data between the TMS320 and the PC in an unsynchronized format; and
5. Program RAM for down-loading and running programs on the TMS320.

The control and status registers enable the PC host to control the TMS320 processor, and to monitor status signals from the TMS320. Control bits include TMS320 reset, processor hold, basic I/O, and interrupt lines. Status bits include processor hold status, external flag status, and the status of the TMS320 reset, basic I/O, and interrupt lines. These registers are mapped into the PC address space, and are accessed as memory locations.

The 16-bit bi-directional communications port is used to pass information between the PC and the TMS320. This port is designed to handle the synchronization problem associated with passing data between two processors with different clock speeds. As a result, the port can be written to, or read from by either the PC, or the TMS320 at any time, and will yield valid data.

The dual-port RAM is also used to pass information between the PC and the TMS320. This memory, however, is not synchronized, and care must be used to ensure that the same location is not simultaneously accessed by both the PC and the TMS320. The communications port can be used to provide information as to which memory locations are being accessed by which processor, and therefore avoid conflict.

The program RAM is used to down-load TMS320 programs from the PC host for subsequent execution by the DSP. This interface block was provided so that programs could be easily updated, and changed to provide different functions for the TMS320.

To digitize the voice samples, a low cost combo CODEC was used. This microelectronic chip provided sampling at 8000 samples per second, and coding into μ -law PCM format, and also

provided anti-alias and reconstruction filtering. This device easily interfaced to the TMS320 processor to transfer the digitized voice data to the DSP.

A standard telephone was used to detect and amplify the speech samples, as well as provide a speaker for voice playback. Interfacing the telephone to the voice digitization and compression system was easily accomplished with the use of a telephone Subscriber Line Interface Circuit (SLIC) integrated circuit.

The following pages contain the complete schematics for the silence compression development system.

Title				UBC Electrical Engineering
Host Interface Plug-in Board				
Size				Document Number
8				PROCHOST.DIA (Chris Rose)
REV				1
Date:				August 20, 1980 (Steel)

UBO Electrical Engineering	
Title	
Processor Board PALs	
Size	Document Number
B	PROCPALs:DIA (Chris Rose)
Date: August 28, 1980 Sheet 5 of 8

UBC Electrical Engineering		
Title		
Analog Interface for Processor Board		
Size	Document Number	REV
B	PROGACQ.DIA (Chris Rose)	1
Date:	August 20, 1990	Sheet 6 of 6

A.2 Firmware Listings

The following pages contain the firmware listings for the programmable logic devices used in the TMS320 DSP system hardware design.

This file contains the listings for the PALS used in the TMS320c25 processor system

HOSTPAL.V2

```

**
** HostPal, Version 2
** U1
**
** Pal to decode the memory address block from the PC host computer
**
DEVICE HOSTPAL (P16L8)
PIN
S1 = 1 (INPUT combinatorial)    "4 switch inputs for segment decoding"
S2 = 2 (INPUT combinatorial)
S3 = 3 (INPUT combinatorial)
S4 = 4 (INPUT combinatorial)
SA16 = 5 (INPUT combinatorial)  "4 MSB address lines for segment decoding"
SA17 = 6 (INPUT combinatorial)
SA18 = 7 (INPUT combinatorial)
SA19 = 8 (INPUT combinatorial)
RESET_DRV = 9 (INPUT combinatorial)    "Reset from PC, active high"
/SMEMW = 11 (INPUT combinatorial)    "PC write request"
/WRITE = 12 (OUTPUT combinatorial active_low)    "Host data latch direction"
/SMEMR = 13 (INPUT combinatorial active_low)    "PC read request"
/PMRD = 14 (IO combinatorial active_low)    "Buffered read request"
/PMWR = 15 (IO combinatorial active_low)    "Buffered write request"
/SEGSEL = 16 (OUTPUT combinatorial active_low)    "Segment selected output"
/T2 = 17 (IO combinatorial active_low)    "2 intermediate lines for"
/T1 = 18 (IO combinatorial active_low)    "segment calculation"
/PRST = 19 (OUTPUT combinatorial active_low)    "Buffered PC reset, act. low"
:
"Logic Equation Section"
BEGIN
T1 = ((/S1) & SA16) * ((/S2) & SA17);
T2 = ((/S3) & SA18) * ((/S4) & SA19);
SEGSEL = (T1 * T2 * /RESET_DRV) + PMWR + PMRD;
WRITE = (SMEMW * T1 * T2 * /RESET_DRV) + PMWR;
PRST = RESET_DRV;
PMWR = SMEMW * T1 * T2 * /RESET_DRV;
PMRD = SMEMR * T1 * T2 * /RESET_DRV;
ENABLE (T1);
ENABLE (T2);
ENABLE (SEGSEL);
ENABLE (WRITE);
ENABLE (PRST);
ENABLE (PMWR);
ENABLE (PMRD);
END.
"Test Vector Section"
TEST VECTORS
IN S1,S2,S3,S4,SA16,SA17,SA18,SA19,RESET_DRV,/SMEMW,/SMEMR;
I_O /T1,/T2,/PMWR,/PMRD;
OUT /SEGSEL,/PRST,/WRITE;
BEGIN
0 1 1 0 0 0 0 0 0 0 0 H H H H H H H;
0 1 1 0 0 0 0 0 1 0 0 1 H H H H H H;
0 1 1 0 0 0 1 0 0 1 0 H L H H H H H;

```

```

0 1 1 0 0 0 1 1 0 1 1 H H H H H H H;
0 1 1 0 0 1 0 0 0 0 0 L H H H H H H;
0 1 1 0 0 1 0 1 0 0 1 L H H H H H H;
0 1 1 0 0 1 1 0 0 1 0 L L H L L H H;
0 1 1 0 0 1 1 1 0 1 1 L H H H H H H;
0 1 1 0 1 0 0 0 0 0 1 H H H H H H H;
0 1 1 0 1 0 0 1 0 1 0 H H H H H H H;
0 1 1 0 1 0 1 0 0 0 1 H L H H H H H;
0 1 1 0 1 0 1 1 0 0 0 H H H H H H H;
0 1 1 0 1 1 0 0 0 1 1 H H H H H H H;
0 1 1 0 1 1 0 1 0 1 0 H H H H H H H;
0 1 1 0 1 1 1 0 0 0 1 H L H H H H H;
0 1 1 0 1 1 1 1 0 0 0 H H H H H H H;
0 1 1 0 0 1 1 0 1 1 0 L L H H H L H;
0 1 1 0 0 1 1 0 0 1 0 L L H L L H H;
0 1 1 0 0 1 1 0 0 0 0 L L L L L H L;
0 1 1 0 0 1 1 0 0 1 1 L L H H L H H;
END.

```

PCDATA.V0

```

**
** PCDATA, Version 0
** U28
**
** Pal to decode accesses by the PC to the dual port data memory
**
DEVICE PCDATA (P16L8)
PIN
PA15 = 1 (INPUT combinatorial)
PA14 = 2 (INPUT combinatorial)
PA13 = 3 (INPUT combinatorial)
PA12 = 4 (INPUT combinatorial)
PA0 = 5 (INPUT combinatorial)
/PMWR = 6 (INPUT combinatorial)
/PMRD = 7 (INPUT combinatorial)
/PLADDR = 12 (OUTPUT combinatorial active_low)
/PCWR = 13 (OUTPUT combinatorial active_low)
/PCRD = 14 (OUTPUT combinatorial active_low)
/PED1H = 15 (OUTPUT combinatorial active_low)
/PED1L = 16 (OUTPUT combinatorial active_low)
/PED0H = 17 (OUTPUT combinatorial active_low)
/PED0L = 18 (OUTPUT combinatorial active_low)
/PDWR = 19 (OUTPUT combinatorial active_low)
:
**
** Logic Equations
**
*****
BEGIN
PDWR = (PMWR * PA15 * (/PA14 * PA13));
PED0L = ((PMWR + PMRD) * PA15 * /PA14 * /PA13 * /PA12 * /PA0);
PED0H = ((PMWR + PMRD) * PA15 * /PA14 * /PA13 * /PA12 * PA0);
PED1L = ((PMWR + PMRD) * PA15 * /PA14 * /PA13 * PA12 * /PA0);
PED1H = ((PMWR + PMRD) * PA15 * /PA14 * /PA13 * PA12 * PA0);
PCRD = (PMRD * PA15 * PA14 * PA13 * /PA12 * /PA0);
PCWR = (PMWR * PA15 * PA14 * PA13 * /PA12 * /PA0);

```

```

PLADDR = ((PMWR + PMRD) * PA15 * PA14 * PA13 * PA12);
ENABLE (PDWR);
ENABLE (PED0L);
ENABLE (PED0H);
ENABLE (PED1L);
ENABLE (PED1H);
ENABLE (PCRD);
ENABLE (PCWR);
ENABLE (PLADDR);
END.
**
** Test vector specification
**
*****
TEST_VECTORS
IN      PA15, PA14, PA13, PA12, PA0, /PMWR, /PMRD;
OUT     /PDWR, /PED0L, /PED0H, /PED1L, /PED1H, /PCRD, /PCWR, /PLADDR;
BEGIN
1 0 0 0 0 0 1 L L H H H H H H;
1 0 0 0 1 0 1 L H L H H H H H;
1 0 0 1 0 0 1 L H H L H H H H;
1 0 0 1 1 0 1 L H H H L H H H;
1 0 1 0 0 0 1 L H H H H H H H;
1 0 1 0 1 0 1 L H H H H H H H;
1 0 1 1 0 0 1 L H H H H H H H;
1 0 1 1 1 0 1 L H H H H H H H;
1 1 0 0 0 0 1 L H H H H H H H;
1 1 0 0 1 0 1 L H H H H H H H;
1 1 0 1 0 0 1 L H H H H H H H;
1 1 0 1 1 0 1 L H H H H H H H;
1 1 1 0 0 0 1 H H H H H H L H;
1 1 1 0 1 0 1 H H H H H H H H;
1 1 1 1 0 0 1 H H H H H H H L;
1 1 1 1 1 0 1 H H H H H H H L;
0 0 0 0 0 0 1 H H H H H H H H;
1 1 1 0 0 1 0 H H H H H L H H;
END.

```

PCPROG.V0

```

**
** PCProg, Version 0
** U29
**
** Pal to control the latches connecting the TMS bus to the PC bus; enabling
** the PC to access the TMS program memory.
**
DEVICE PCPROG (P16L8)
PIN
/PLADDR = 1      (INPUT combinatorial)
PA15    = 2      (INPUT combinatorial)
PA0      = 3      (INPUT combinatorial)
/PMWR    = 4      (INPUT combinatorial)
/PMRD    = 5      (INPUT combinatorial)
/THOLDA  = 6      (INPUT combinatorial)
PLIE     = 7      (INPUT combinatorial)
/SYNCHW  = 8      (INPUT combinatorial)
/PPADDR  = 12     (OUTPUT combinatorial active_low)

```

```

PLWH     = 13     (OUTPUT combinatorial active_low)
PLWL     = 14     (OUTPUT combinatorial active_low)
/PLRH    = 15     (OUTPUT combinatorial active_low)
/PLRL    = 16     (OUTPUT combinatorial active_low)
/PPDH    = 17     (OUTPUT combinatorial active_low)
/PPDL    = 18     (OUTPUT combinatorial active_low)
/PPWR    = 19     (OUTPUT combinatorial active_low)
;
**

```

** Logic Equations

```

*****
BEGIN
PPADDR = (THOLDA * (PMWR + PMRD) * /PA15);
PPWR = (THOLDA * PMWR * /PA15);
PPDL = (THOLDA * (PMWR + PMRD) * /PA15 * /PA0);
PPDH = (THOLDA * (PMWR + PMRD) * /PA15 * PA0);
PLRL = (PLADDR * /PLIE * PMRD * /PA0);
PLRH = (PLADDR * /PLIE * PMRD * PA0);
/PLWL = (PLADDR * SYNCHW * PMWR * /PA0);
/PLWH = (PLADDR * SYNCHW * PMWR * PA0);
ENABLE (PPADDR);
ENABLE (PPWR);
ENABLE (PPDL);
ENABLE (PPDH);
ENABLE (PLRL);
ENABLE (PLRH);
ENABLE (PLWL);
ENABLE (PLWH);
END.
**

```

** Test Vector Specification

```

*****
TEST_VECTORS
IN      PA15, PA0, /PMWR, /PMRD, /THOLDA, /PLADDR, PLIE, /SYNCHW;
OUT     /PPADDR, /PPWR, /PPDL, /PPDH, /PLRL, /PLRH, PLWL, PLWH;
BEGIN
0 0 0 1 0 1 1 1 L L L H H H L L;
0 1 0 1 0 1 1 1 L L H L H H L L;
0 0 1 0 0 1 1 1 L H L H H H L L;
0 1 1 0 0 1 1 1 L H H L H H L L;
0 0 0 1 1 1 1 1 H H H H H H L L;
1 0 0 1 0 1 1 1 H H H H H H L L;
1 0 1 0 1 0 0 1 H H H H L H L L;
1 1 1 0 1 0 0 1 H H H H H L L L;
1 1 1 0 1 0 1 1 H H H H H H L L;
1 0 0 1 1 0 1 0 H H H H H H L L;
1 1 0 1 1 0 1 0 H H H H H H L H;
1 1 0 1 1 0 1 1 H H H H H H L L;
END.

```

TMSDATA.V0

```

**
** TMSData, Version 0
** U31
**

```

```
** Pal to decode the accesses by the TMS320 to the dual port data memory
```

```
**
```

```
DEVICE TMSDATA (P16L8)
```

```
PIN
```

```
TA15 = 1 (INPUT combinatorial)
TA14 = 2 (INPUT combinatorial)
TA13 = 3 (INPUT combinatorial)
TA12 = 4 (INPUT combinatorial)
TA11 = 5 (INPUT combinatorial)
/TDS = 6 (INPUT combinatorial)
/TWR = 7 (INPUT combinatorial)
/TSTRB = 8 (INPUT combinatorial)
/TMSC = 9 (INPUT combinatorial)
/TRST = 11 (INPUT combinatorial)
/TIS = 13 (INPUT combinatorial active_low)
/TLRD = 14 (OUTPUT combinatorial active_low)
/TLWR = 15 (OUTPUT combinatorial active_low)
/TREADY = 16 (OUTPUT combinatorial active_low)
/TED1 = 17 (OUTPUT combinatorial active_low)
/TED0 = 18 (OUTPUT combinatorial active_low)
/TDWR = 19 (OUTPUT combinatorial active_low)
```

```
;
```

```
**
```

```
** Logic Equations
```

```
**
```

```
*****
```

```
BEGIN
```

```
TDWR = (TDS * TWR * TSTRB * /TRST * /TA15 * /TA14 * (TA13 + TA12));
```

```
TED0 = (TDS * /TA15 * /TA14 * /TA13 * TA12 * /TA11);
```

```
TED1 = (TDS * /TA15 * /TA14 * /TA13 * TA12 * TA11);
```

```
/TREADY = (/TDS + TA15 + TA14 + (/TA13 * /TA12) + /TMSC);
```

```
TLWR = (TIS * TSTRB * TWR * /TRST);
```

```
TLRD = (TIS * /TWR);
```

```
ENABLE (TDWR);
```

```
ENABLE (TED0);
```

```
ENABLE (TED1);
```

```
ENABLE (TREADY);
```

```
ENABLE (TLWR);
```

```
ENABLE (TLRD);
```

```
END.
```

```
**
```

```
** Test Vector Specification
```

```
**
```

```
*****
```

```
TEST_VECTORS
```

```
IN TA15, TA14, TA13, TA12, TA11, /TDS, /TWR, /TSTRB, /TMSC, /TRST, /TIS;
```

```
OUT /TDWR, /TED0, /TED1, TREADY, /TLWR, /TLRD;
```

```
BEGIN
```

```
0 0 0 0 0 0 0 0 0 1 1 H H H H H H;
```

```
0 0 0 0 1 0 0 0 0 1 1 H H H H H H;
```

```
0 0 0 1 0 0 0 0 0 1 1 L L H L H H;
```

```
0 0 0 1 1 0 0 0 0 1 1 L H L L H H;
```

```
0 0 1 0 0 0 0 0 0 1 1 L H H L H H;
```

```
0 0 1 0 1 0 0 0 0 1 1 L H H L H H;
```

```
0 0 1 1 0 0 0 0 0 1 1 L H H L H H;
```

```
0 0 1 1 1 0 0 0 0 1 1 L H H L H H;
```

```
0 1 0 0 0 0 0 0 0 1 1 H H H H H H;
```

```
0 1 0 0 1 0 0 0 0 1 1 H H H H H H;
```

```
0 1 0 1 0 0 0 0 0 1 1 H H H H H H;
```

```
0 1 0 1 1 0 0 0 0 1 1 H H H H H H;
```

```
0 1 1 0 0 0 0 0 0 1 1 H H H H H H;
```

```
0 1 1 0 1 0 0 0 0 1 1 H H H H H H;
```

```
0 1 1 1 0 0 0 0 0 1 1 H H H H H H;
```

```
0 1 1 1 1 0 0 0 0 1 1 H H H H H H;
```

```
1 0 0 0 0 0 0 0 0 1 1 H H H H H H;
```

```
0 0 0 0 0 1 0 0 0 1 0 H H H H L H;
```

```
0 0 0 0 0 1 1 1 0 1 0 H H H H H L;
```

```
0 0 0 1 0 0 0 0 0 1 1 1 L L H H H H;
```

```
END.
```

PROGMEM.V0

```
**
```

```
** ProgMem, Version 0
```

```
** U30
```

```
**
```

```
** Pal to control the enable lines on the actual program memory
```

```
**
```

```
DEVICE PROGEM (P16L8)
```

```
PIN
```

```
/PPADDR = 1 (INPUT combinatorial)
PA0 = 2 (INPUT combinatorial)
/PMWR = 3 (INPUT combinatorial)
/PMRD = 4 (INPUT combinatorial)
/TPS = 5 (INPUT combinatorial)
/TWR = 6 (INPUT combinatorial)
/TSTRB = 7 (INPUT combinatorial)
TA15 = 8 (INPUT combinatorial)
TA14 = 9 (INPUT combinatorial)
TA13 = 11 (INPUT combinatorial)
/FFSET = 12 (OUTPUT combinatorial active_low)
/PLADDR = 13 (INPUT combinatorial active_low)
/TRST = 14 (IO combinatorial active_low)
/PRST = 15 (INPUT combinatorial active_low)
/SRST = 16 (INPUT combinatorial active_low)
/WRHP = 17 (OUTPUT combinatorial active_low)
/WRLP = 18 (OUTPUT combinatorial active_low)
/ENP0 = 19 (OUTPUT combinatorial active_low)
```

```
;
```

```
**
```

```
** Logic Equations
```

```
**
```

```
*****
```

```
BEGIN
```

```
ENP0 = (((PPADDR * (PMWR + PMRD)) + TPS) * /TA15 * /TA14 * /TA13);
```

```
WRLP = ((PPADDR * PMWR * /PA0) + (TPS * TWR * TSTRB * /TRST));
```

```
WRHP = ((PPADDR * PMWR * PA0) + (TPS * TWR * TSTRB * /TRST));
```

```
FFSET = (/PLADDR + TRST);
```

```
TRST = (PRST + SRST);
```

```
ENABLE (ENP0);
```

```
ENABLE (WRLP);
```

```
ENABLE (WRHP);
```

```
ENABLE (FFSET);
```

```
ENABLE (TRST);
```

```
END.
```

```
**
```

```
** Test Vector Specification
```

```
**
```

TEST_VECTORS

IN TA15, TA14, TA13, PA0, /PPADDR, /PMWR, /PMRD, /TPS, /TWR, /TSTRB,
/PLADDR, /PRST, /SRST;

IO /TRST;

OUT /ENPG, /WRLP, /WRHP, /FFSET;

BEGIN

0 0 0 0 0 0 1 1 1 1 1 1 1 H L L H L;

0 0 0 1 0 0 1 1 1 1 1 1 1 H L H L L;

0 0 0 0 0 1 0 1 1 1 1 1 1 H L H H L;

0 0 0 1 0 1 0 1 1 1 1 1 1 H L H H L;

0 0 0 0 1 1 1 0 0 0 1 1 1 H L L L L;

0 0 0 1 1 1 1 0 0 0 1 1 1 H L L L L;

0 0 0 0 1 1 1 0 1 1 1 1 1 H L H H L;

0 0 0 1 1 1 1 0 1 1 1 1 1 H L H H L;

1 0 0 0 0 0 1 1 1 1 1 1 1 H H L H L;

1 0 0 0 1 1 1 0 0 0 0 1 1 H H L L H;

1 0 0 0 1 1 1 1 1 1 1 0 1 L H H H L;

1 0 0 0 1 1 1 1 1 1 0 0 1 L H H H L;

1 0 0 0 1 1 1 1 1 1 1 1 0 L H H H L;

END.

Appendix B

Software Listings

B.1 Simulation Listings

The development of the silence detection algorithm relied heavily on the use of a software simulator. This enabled the use of identical speech samples to be used on various derivatives of the silence detection algorithm, enabling meaningful comparisons between the various compression statistics.

The software simulator operated on a previously stored speech file, removing the silence intervals and created a new output speech file as well as the compression statistics obtained. Playback of the compressed speech file enabled evaluations of speech quality to be made.

The following pages contain the listings of the C language programs that comprise the software simulator for the silence detection algorithm.

This file contains the program used to simulate the silence compression algorithm on an IBM-PC/AT.

SilSim.h

```

/*
** SilSim.h
** March 27, 1990
** Chris Rose
**
** Header file for silence deletion simulation program.
*/

/* Default parameters */
#define FRAMESIZE      16      /* Number of samples to calc. energy, etc */
#define MINTHRESH      2.0     /* Threshold for speech */
#define MAXTHRESH      2.5
#define ZEROCROSS      8       /* Zero-crossings threshold */
#define MINSPEECH      10
#define MINSIL         4
#define HANGOVER       20
#define ECRIT          1.28
#define SPCAVGBLKLEN   32
#define SPCAVGNUMBLK   32
#define SILAVGLEN      64
#define INITSPPECHAVG  500
#define INITSLAVG      20
#define SPEECHSCALE    20
#define INITSCALE      20

#define SILCODE        0x0080
#define ZERO           0xFF

typedef enum    { NONE, NOISE, BOTH } AdaptType;
typedef enum    { NO, YES } Query;

/* Global variable declaration */
extern int      FrameSize, ZeroCross, MinSpeech, MinSil, HangOver;
extern int      SpcAvgBlkLen, SilAvgLen, InitSilAvg, InitSpeechAvg;
extern int      SpeechScale, InitScale, SpcAvgNumBlk;
extern double   MinThresh, MaxThresh, ECrit;
extern AdaptType Adaptation;
extern Query     NoiseIn, Trace, ECritOn;

extern char      **Frame;
extern int       MaxValue, MinValue;
extern unsigned  MaxEnergy, MinEnergy;
extern unsigned  NumFrames, SilFrames, SpeechFrames;
extern unsigned  SilIntervals, SpeechIntervals;
extern unsigned  ZFrames, EFrames, BFrames, NFrames;

```

SilSim.c

```

/*
** SilSim.c
** March 26, 1990

```

```

** Chris Rose
**
** Program to perform silence deletion on a speech data file.
*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include "silsim.h"

/* Global variable declaration */
int      FrameSize, ZeroCross, MinSpeech, MinSil, HangOver;
int      SpcAvgBlkLen, SilAvgLen, InitSilAvg, InitSpeechAvg;
int      SpeechScale, InitScale, SpcAvgNumBlk;
double   MinThresh, MaxThresh, ECrit;
AdaptType Adaptation;
Query     NoiseIn, Trace, ECritOn;

char      **Frame;
int       MaxValue, MinValue;
unsigned  MaxEnergy, MinEnergy;
unsigned  NumFrames, SilFrames, SpeechFrames;
unsigned  SilIntervals, SpeechIntervals;
unsigned  ZFrames, EFrames, BFrames, NFrames;

```

```

/* Function Prototypes */
char **GetMatrix( int Rows, int Columns );
void  FreeMatrix( char **Frame );
void  DeleteSilence( FILE *InFile, FILE *OutFile );

```

```

/*****
void main( int Argc, char *Argv[] )
*****/
/*****
{
    FILE      *InFile, *OutFile;
    char      *InFileName, *Str;
    int       StoreFrames;
    double    PercentSil, PercentSpeech, AvgSilLength, AvgSpeechLength;
    long      NumCodeBytes, NumSpeechBytes, TotalBytes;
    double    Compression, PercentOH;

```

```

/* Set default values */
FrameSize      = FRAMESIZE;
MinThresh      = MINTHRESH;
MaxThresh      = MAXTHRESH;
ZeroCross      = ZEROCROSS;
MinSpeech      = MINSPEECH;
MinSil         = MINSIL;
HangOver       = HANGOVER;
SilAvgLen      = SILAVGLEN;
SpcAvgBlkLen   = SPCAVGBLKLEN;
SpcAvgNumBlk   = SPCAVGNUMBLK;
InitSpeechAvg  = INITSPPECHAVG;
InitSilAvg     = INITSLAVG;
ECrit          = ECRIT;
SpeechScale    = SPEECHSCALE;
InitScale      = INITSCALE;
Adaptation     = BOTH;
NoiseIn        = YES;
Trace          = NO;
ECritOn        = YES;

```

```

if (Argc < 2) {
    fprintf(stderr, "Usage: SILSIM <OutFile>");
    exit(EXIT_FAILURE);
}

/* Get environment variable options */
if ((InFileName = getenv("INFILE")) == NULL) {
    fprintf(stderr, "Error: Cannot get input file-spec");
    exit(EXIT_FAILURE);
}

if ((Str = getenv("FRAME SIZE")) != NULL) FrameSize = atoi(Str);
if ((Str = getenv("MIN THRESH")) != NULL) MinThresh = atof(Str);
if ((Str = getenv("MAX THRESH")) != NULL) MaxThresh = atof(Str);
if ((Str = getenv("ZERO CROSS")) != NULL) ZeroCross = atoi(Str);
if ((Str = getenv("MIN SPEECH")) != NULL) MinSpeech = atoi(Str);
if ((Str = getenv("MIN SIL")) != NULL) MinSil = atoi(Str);
if ((Str = getenv("HANG OVER")) != NULL) HangOver = atoi(Str);
if ((Str = getenv("ECRIT")) != NULL) ECRIT = atoi(Str);
if ((Str = getenv("SIL AVG LEN")) != NULL) SilAvgLen = atoi(Str);
if ((Str = getenv("SPCAVG BLK LEN")) != NULL) SpcAvgBlkLen = atoi(Str);
if ((Str = getenv("SPCAVG NUM BLK")) != NULL) SpcAvgNumBlk = atoi(Str);
if ((Str = getenv("INIT SPEECH AVG")) != NULL) InitSpeechAvg = atoi(Str);
if ((Str = getenv("INIT SIL AVG")) != NULL) InitSilAvg = atoi(Str);
if ((Str = getenv("SPEECH SCALE")) != NULL) SpeechScale = atoi(Str);
if ((Str = getenv("INIT SCALE")) != NULL) InitScale = atoi(Str);
if ((Str = getenv("ADAPTATION")) != NULL) Adaptation = atoi(Str);
if ((Str = getenv("NOISE IN")) != NULL) NoiseIn = atoi(Str);
if ((Str = getenv("TRACE")) != NULL) Trace = atoi(Str);
if ((Str = getenv("ECRIT ON")) != NULL) ECRITon = atoi(Str);

if ((InFile = fopen(InFileName, "rb")) == NULL) {
    fprintf(stderr, "Error: Cannot open file \"%s\"", InFileName);
    exit(EXIT_FAILURE);
}

if ((OutFile = fopen(Argv[1], "wb")) == NULL) {
    fprintf(stderr, "Error: Cannot open file \"%s\"", Argv[1]);
    exit(EXIT_FAILURE);
}

StoreFrames = max(MinSpeech, MinSil);
if ((Frame = GetMatrix(StoreFrames, FrameSize)) == NULL) {
    fprintf(stderr, "Error: Cannot allocate FRAME matrix");
    exit(EXIT_FAILURE);
}

/* Perform the actual silence removal */
DeleteSilence(InFile, OutFile);

FreeMatrix(Frame);
fclose(InFile);
fclose(OutFile);

/* Display the compression results */
PercentSil = 100.0*SilFrames/NumFrames;
PercentSpeech = 100.0*SpeechFrames/NumFrames;
AvgSilLength = ((double)SilFrames)/SilIntervals;
AvgSpeechLength = ((double)SpeechFrames)/SpeechIntervals;
TotalBytes = (long) FrameSize * NumFrames;

```

```

NumCodeBytes = ((long) SilIntervals * 6) + 4;
NumSpeechBytes = (long) FrameSize * SpeechFrames;
Compression = 100.0*(NumCodeBytes+NumSpeechBytes)/TotalBytes;
PercentOH = 100.0*NumCodeBytes/TotalBytes;

printf("\nAlgorithm Parameters:\n");
printf(" FrameSize = %d, MinThresh = %4.2lf, ", FrameSize, MinThresh);
printf("MaxThresh = %4.2lf, ZeroCross = %d\n", MaxThresh, ZeroCross);
printf(" MinSpeech = %d, MinSil = %d, ", MinSpeech, MinSil);
printf("HangOver = %d\n", HangOver);

printf("\nSpeech File:\n");
printf(" MinValue = %d, MaxValue = %d, ", MinValue, MaxValue);
printf("MinEnergy = %u, MaxEnergy = %u\n", MinEnergy, MaxEnergy);
printf(" NumFrames = %u, NumBytes = %ld\n", NumFrames, TotalBytes);

printf("\nAlgorithm Results:\n");
printf(" SilFrames = %u (%5.2lf%) ", SilFrames, PercentSil);
printf("(%u), ", ((unsigned long)SilFrames*FrameSize));
printf("SpeechFrames = %u (%5.2lf%) ", SpeechFrames, PercentSpeech);
printf("(%u)\n", ((unsigned long)SpeechFrames*FrameSize));
printf(" Overhead = %5.2lf% ", PercentOH);
printf("(%u), ", NumCodeBytes);
printf("Percent File Compression = %5.2lf%\n", Compression);
printf(" SilIntervals = %u, ", SilIntervals);
printf("SpeechIntervals = %u\n", SpeechIntervals);
printf(" AvgSilLength = %5.2lf ", AvgSilLength);
printf("(%5.1lf), ", AvgSilLength*FrameSize);
printf("AvgSpeechLength = %5.2lf ", AvgSpeechLength);
printf("(%5.1lf)\n", AvgSpeechLength*FrameSize);
printf(" ZFrames = %u, EFrames = %u, ", ZFrames, EFrames);
printf("BFrames = %u, NFrames = %u\n", BFrames, NFrames);

exit(EXIT_SUCCESS);

```

NoiseDel.c

```

/*
** NoiseDel.c
** May 18, 1990
** Chris Rose
**
** Routines to perform the actual silence deletion on the speech sample.
** This version will substitute gaussian noise in the silence intervals.
**

#include <stdio.h>
#include <stdlib.h>
#include "silsim.h"

typedef union {
    unsigned w;
    char b[2];
} DATATYPE;

typedef enum { SPEECH, SILENCE } StateType;

```



```

int MuLawExpand( int MuLawCode );
void SaveSilenceFrame( int AvgEnergy, FILE *OutFile );
void SaveSpeechFrame( int FrameNumber, FILE *OutFile );
int MuLawCompress( int LinearCode );
int GenerateGauss( int Avg );

/*****
void DeleteSilence( FILE *InFile, FILE *OutFile )
*****/
{
    int      Index, Energy, LinValue, CurThresh, Previous, Crossings;
    int      FrameCount, StoreLevel;
    unsigned SillLength, SpeechLength;
    long     EnergySum, AvgSum;
    DATAWORD CurWord;
    StateType State, CurrentFrame;
    int      AvgList[NUMAVG], Avg, AvgPoint;

    srand(12345);

    for (Index = 0; Index < NUMAVG; Index++) {
        AvgList[Index] = INITAVG;
    }
    AvgSum = INITAVG*NUMAVG;
    Avg = INITAVG;
    AvgPoint = 0;

    MaxValue = MinValue = MaxEnergy = 0;
    MinEnergy = 1000;
    NumFrames = SilFrames = SpeechFrames = 0;
    SilIntervals = SpeechIntervals = 0;

    SillLength = SpeechLength = 0;
    StoreLevel = 0;
    State = SILENCE;
    CurThresh = MaxThresh;
    Previous = 0;

    while (1) {

        /* Calculate the frame energy, and zero-crossings */
        FrameCount = Crossings = 0;
        EnergySum = 0L;
        while (FrameCount < FrameSize) {

            /* Get next two bytes from the file */
            for (Index = 0; Index < 2; Index++) {
                CurWord.b[Index] = getc(InFile);
            }

            /* Check for an End Of Data code, if present, stop reading */
            if (CurWord.w == SILCODE) break;

            /* Add these two bytes components to energy and crossings */
            for (Index = 0; Index < 2; Index++) {
                Frame[StoreLevel][FrameCount++] = CurWord.b[Index];
                LinValue = MuLawExpand((int) CurWord.b[Index]);
                EnergySum += (long) abs(LinValue);
                if ((LinValue ^ Previous) & 0x8000) Crossings++;
                Previous = LinValue;
            }
        }
    }
}

```

```

        if (LinValue > MaxValue) MaxValue = LinValue;
        if (LinValue < MinValue) MinValue = LinValue;
    }

    /* Check if end of data was detected */
    if (CurWord.w == SILCODE) break;

    /* Calculate frame energy average */
    Energy = (int)(EnergySum / FrameSize);
    if (Energy > MaxEnergy) MaxEnergy = Energy;
    if (Energy < MinEnergy) MinEnergy = Energy;

    /* Check if frame was speech or silence */
    if ((Energy >= CurThresh) || (Crossings >= ZeroCross)) {
        CurrentFrame = SPEECH;
    }
    else {
        CurrentFrame = SILENCE;
    }

    if ((State == SILENCE) && (CurrentFrame == SILENCE)) {
        for (Index = 0; Index < (SpeechLength+1); Index++) {
            if (Energy < (Avg * AVGCRT)) {
                AvgSum -= AvgList[AvgPoint];
                AvgSum += Energy;
                Avg = (int) AvgSum/NUMAVG;
                AvgList[AvgPoint++] = Energy;
                if (AvgPoint >= NUMAVG) AvgPoint = 0;
            }
            SaveSilenceFrame(Avg, OutFile);
            NumFrames++; SilFrames++;
            SillLength++;
        }
        if (SillLength >= HangOver) CurThresh = MaxThresh;
        SpeechLength = StoreLevel = 0;
    }
    else if ((State == SILENCE) && (CurrentFrame == SPEECH)) {
        CurThresh = MinThresh;
        SpeechLength++;
        if (SpeechLength == MinSpeech) {
            State = SPEECH;
            for (Index = 0; Index < SpeechLength; Index++) {
                SaveSpeechFrame(Index, OutFile);
                NumFrames++; SpeechFrames++;
            }
            if (SillLength > 0) SilIntervals++;
            SillLength = StoreLevel = 0;
        }
        else {
            StoreLevel++;
        }
    }
    else if ((State == SPEECH) && (CurrentFrame == SPEECH)) {
        CurThresh = MinThresh;
        for (Index = 0; Index < (SillLength+1); Index++) {
            SaveSpeechFrame(Index, OutFile);
            NumFrames++; SpeechFrames++;
            SpeechLength++;
        }
    }
}

```

```

        }
        SillLength = StoreLevel = 0;
    }
    else { /* State = SPEECH, CurrentFrame = Silence */
        SillLength++;
        if (SillLength >= HangOver) CurThresh = MaxThresh;
        if (SillLength == MinSil) {
            State = SILENCE;
            if (Energy < (Avg * AVGCRT)) {
                AvgSum -= AvgList[AvgPoint];
                AvgSum += Energy;
                Avg = (int) AvgSum/NUMAVG;
                AvgList[AvgPoint++] = Energy;
                if (AvgPoint >= NUMAVG) AvgPoint = 0;
            }
            for (Index = 0; Index < SillLength; Index++) {
                SaveSilenceFrame(Avg, OutFile);
                NumFrames++; SilFrames++;
            }
            if (SpeechLength > 0) SpeechIntervals++;
            SpeechLength = StoreLevel = 0;
        }
        else {
            StoreLevel++;
        }
    }
}

/* Save any stored frames of speech or silence */
if (State == SILENCE) {
    for (Index = 0; Index < SpeechLength; Index++) {
        SaveSilenceFrame(Avg, OutFile);
    }
}
else if (State == SPEECH) {
    for (Index = 0; Index < SillLength; Index++) {
        SaveSpeechFrame(Index, OutFile);
    }
}

/* Mark end of speech file */
putw(SILCODE, OutFile);
}

/*****
void SaveSilenceFrame( int Avg, FILE *OutFile )
*****/
{
    int    Index, MuLawCode, LinearCode;

    LinearCode = GenerateGauss(Avg);
    MuLawCode = MuLawCompress(LinearCode);
    for (Index = 0; Index < FrameSize; Index++) {
        putc(MuLawCode, OutFile);
    }
}

/*****
void SaveSpeechFrame( int FrameNumber, FILE *OutFile )
*****/

```

```

{
    int    Index;

    for (Index = 0; Index < FrameSize; Index++) {
        putc(Frame[FrameNumber][Index], OutFile);
    }
}

```

uLawExp.c

```

/*
** uLawExp.c
** March 16, 1990
** Chris Rose
**
** Routine to perform Mu-law expansion in C-code.
*/

#define BIAS      33

/*****
int MuLawExpand( int MuLawCode )
*****/
{
    int Sign, LinSeg, Quant, LinVal, CompVal;

    CompVal = ~MuLawCode;
    Sign = (CompVal & 0x0080);
    LinSeg = ((CompVal & 0x0070) >> 4);
    Quant = (CompVal & 0x000F);

    Quant = ((Quant * 2) + BIAS);
    LinVal = (Quant << LinSeg);
    LinVal -= BIAS;

    if (Sign == 0) {
        return(LinVal);
    }
    else {
        return(-LinVal);
    }
}

```

uLawComp.c

```

/*
** uLawComp.c
** March 20, 1990
** Chris Rose
**
** Routine to perform Mu-law compression.
*/

#define BIAS 33

/*****

```

```

int MuLawCompress( int LinearCode )
/*****
{
    int ShiftCount, Quant, LinSeg, Compressed;

    ShiftCount = 0;
    if (LinearCode < 0) {
        LinearCode = -LinearCode + BIAS;
        /* Check for overflow of 13 bit magnitude */
        if ((LinearCode & 0xE000) > 0) LinearCode = 0x1FFF;
        while ((ShiftCount < 7) && ((LinearCode & 0x1000) == 0)) {
            LinearCode <<= 1;
            ShiftCount++;
        }
        LinSeg = 7 - ShiftCount;
        Quant = (LinearCode & 0x0F00) >> 8;
        Compressed = (~(LinSeg << 4) | Quant) & 0x007F;
    }
    else {
        LinearCode += BIAS;
        /* Check for overflow of 13 bit magnitude */
        if ((LinearCode & 0xE000) > 0) LinearCode = 0x1FFF;
        while ((ShiftCount < 7) && ((LinearCode & 0x1000) == 0)) {
            LinearCode <<= 1;
            ShiftCount++;
        }
        LinSeg = 7 - ShiftCount;
        Quant = (LinearCode & 0x0F00) >> 8;
        Compressed = (~(LinSeg << 4) | Quant) & 0x00FF;
    }

    return(Compressed);
}

```

Gauss.c

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>

double      Norm( void );
double      Uniform( void );

double      Seed = 0.0;

int GenerateNoise( int Avg )
{
    double Number;

    Number = Avg * 1.248 * Norm();
    return((int) Number);
}

/*
** Norm
** This routine generates a roughly Gaussian random number that has zero
** mean, and variance of 1 (the normal distribution). The method is based

```

```

** on the "central limit theorem" in the sum of independantly distributed
** random variables approaches the gaussian distribution in the limit. Here,
** 12 'independant' uniform numbers are added, which gives a unit variance,
** but a mean of 6. Therefore, 6 is subtracted from the result to give a
** zero mean. This method is used as it uses only simple arithmetic
** operations; no "sine" or "logarithmic" transformations are necessary.
*/
/*****
double Norm( void )
/*****
{
    int      i;
    double sum;

    sum = -6.0;
    for (i=0; i<12; i++) {
        sum += Uniform();
    }
    return (sum);
}

double Uniform( void )
{
    Seed = fmod(Seed*67081293.0 + 14181771.0, 67108864.0);
    return(Seed/67108864.0);
}

```

Matrix.c

```

/*
** Matrix.c
** March 27, 1990
** Chris Rose
**
** Routines to perform dynamic allocation and de-allocation of matrices.
*/

#include <stdio.h>
#include <stdlib.h>

void AllocError( char *String );

/*****
char **GetMatrix( int Rows, int Columns )
/*****
{
    int      i;
    char      **Matrix;

    /* Allocate row pointers plus 1 extra for NULL */
    Matrix = (char **) malloc((size_t) ((Rows + 1) * sizeof(char*)));
    if (Matrix == NULL) AllocError("Allocation failure 1 in GetMatrix");

    /* Allocate rows, and initialize pointers to them */
    for (i = 0; i < Rows; i++) {
        Matrix[i] = (char *) malloc((size_t) (Columns * sizeof(char)));
        if (Matrix[i] == NULL) {
            AllocError("Allocation failure 2 in GetMatrix");

```

```

    }
    /* Terminate Row pointers with a NULL pointer */
    Matrix[Rows] = NULL;

    /* Return pointer to array of row pointers */
    return(Matrix);
}

/*****
void FreeMatrix( char **Matrix )
*****/
{
    int i;

    i = 0;
    /* Free matrix rows, until NULL pointer is found (after last row) */
    while (Matrix[i] != NULL) {
        free((char *) Matrix[i++]);
    }
    free((char *) Matrix);
}

/*****
void AllocError( char *String )
*****/
{
    fprintf(stderr, "Run Time Allocation Error: %s\n", String);
    exit(EXIT_FAILURE);
}

```

B.2 Real-time Implementation Listings

The real-time implementation of the speech compression algorithm required two basic divisions of programs; the code for the PC host, and the code for the TMS320 DSP system. The TMS320 DSP system program was written in TMS320c25 assembler language and implements the entire silence compression algorithm. The PC host program was written in the C language, and implements the hard disk transfers that are essential to the operation of a real-time speech storage system.

B.2.1 Tms320c25 Program Listings

The following pages contain the listings of the Tms320c25 assembler language code to implement the real-time silence compression algorithm.

The routines to convert between μ -law coded PCM speech and linear PCM for computation of short-term statistics were modified from [38]. The routines to convert between linear PCM and 4-bit ADPCM were modified from [32].

This file contains the TMS320c25 routines that implement the silence compression and playback algorithms.

SilRec.inc

```

;*
;* SilRec.inc
;* May 1, 1990
;* Chris Rose
;*
;* Include file for SilRec modules. This file contains general defines that
;* are used in the Silence Deletion algorithm.
;*****
;*
;* Memory mapped registers
DRR      .set    0          ; Serial port data receive register
DXR      .set    1          ; Serial port data transmit register
TIM      .set    2          ; Timer register
PRD      .set    3          ; Period register
IMR      .set    4          ; Interrupt mask register
GREG     .set    5          ; Global memory allocation register
;*
;* Hardware configuration constants
IntrptMask .set    0010h    ; Enables RINT only (Serial port receive).
CommLatch  .set    0        ; I/O port for Communication Latch to PC.
DataOffset .set    1000h    ; Offset to start of external data memory
DataMemSize .set    1000h   ; Size (in words) of external data memory
;*
;* Silence deletion algorithm constants
SilCodeWord .set    0080h   ; Code word to signal silence frame
EndCodeWord .set    0FFFFh  ; End of data code word
MaxSil      .set    0FFFEh  ; Maximum number of frames of silence stored
;*
;* Silence deletion algorithm parameters
FrameSize   .set    16      ; The number of speech samples per frame
FSShift     .set    4        ; Number of shifts to divide by FrameSize
HighThresh  .set    3B86h    ; High energy threshold (3.72)
LowThresh   .set    319Ah    ; Low energy threshold (3.10)
ZCTHresh    .set    200      ; Zero crossing threshold
MinSpeech   .set    1        ; Minimum number of frames of speech stored
MinSil      .set    1        ; Minimum number of frames of silence stored
LTHangOver  .set    1        ; Number of silence frames until EThresh=High
;*
;* Adaptation constants
REGPAGE     .set    0        ;
SPEECHPAGE  .set    6        ; Speech Avg variables are on page 6
SILPAGE     .set    7        ; Silence Avg variables are on page 7
SPCBLKLEN   .set    16       ; Number of frames per averaging block
SPCAVGSFT   .set    4        ; Shift count to divide by SPCBlkLen
SPCNUMBLK   .set    32       ; Number of blocks of averages for Speech
SPCBLKSFT   .set    5        ; Shift count to divide by SPCNumBlk
SILAVGLEN   .set    64       ; Number of frames for averaging Silence
SILAVGSFT   .set    6        ; Shift count to divide by SilAvgLen
INITSPCAVG  .set    250      ; Initial value for speech average
INITSILAVG  .set    -1       ; Initial value for silence average
CECRIT      .set    147Bh    ; 1.28
CSPSCALE    .set    13
INITSCALE   .set    20

```

```

;*
;* Define Auxiliary Register Variables
TMP      .set    AR0        ; Temporary pointer for general use
CDP      .set    AR1        ; Current Data Pointer
NDP      .set    AR2        ; New Data Pointer
FSC      .set    AR3        ; Frame Samples Counter
ZCR      .set    AR4        ; Zero crossings running counter
SIL      .set    AR5        ; Number of consecutive silence intervals
SPC      .set    AR6        ; Number of consecutive speech intervals
AVG      .set    AR7        ; Pointer to average silence/speech arrays
;*
;* Auxiliary Register Uses:
;* AR0 - (TMP) Used as a temporary pointer. Values in this register are
;* not guaranteed outside of "local" code. This register will also
;* be used by subroutines such as MuLawExp and MuLawComp.
;* AR1 - (CDP) Current Data Pointer. Used to address the current data
;* storage location in external data memory.
;* AR2 - (NDP) New Data Pointer. Points to the first data location that
;* has not yet been verified. Any previous data locations are ready
;* to be transferred to the PC.
;* AR3 - (FSC) Frame Sample Count; Counts down the remaining number of
;* samples required until the frame is completed.
;* AR4 - (ZCR) Zero Crossing Rate; Counts the number of zero-crossings
;* present in a frame of speech samples.
;* AR5 - (SIL) Silence Interval Length; Counts the number of consecutive
;* silence frames encountered in the current silence interval.
;* AR6 - (SPC) Speech Interval Length; Counts the number of consecutive
;* speech frames encountered in the current speech interval.
;* AR7 - (ASE) Average Silence Energy pointer; Points to the first entry
;* to be over-written in the table of most recent silence intervals
;* average energy values.

```

SilRec.asm

```

;*
;* SilRec.asm
;* May 17, 1990
;* Chris Rose
;*
;* This file contains the main routine for the Mu-law silence compressed
;* record program.
;*****
;*
;* Include defines file
;include "silrec.inc"
;*
;* External references
.ref    InitTms320, InitDataTransfer, InitMuLawExp, InitSilenceCompress
.ref    GetSpeechFrame, EvaluateFrame, ProcessFrame
.ref    WriteEndCode, Receive, Unused, InitAvgs
;*
;*****
;sect    "IRUPTS"
;*****
;* Interrupt Vectors
RsVect:  B      Start      ; Reset vector
IOVect:  B      Unused     ; Hardware interrupt 0 vector
IIVect:  B      Unused     ; Hardware interrupt 1 vector

```

```

I2Vect:  B      Unused    ; Hardware interrupt 2 vector
.space 100h                ; reserved interrupt vector space
TmVect:  B      Unused    ; Internal timer interrupt vector
RxVect:  B      Receive   ; Serial port receive interrupt vector
TxVect:  B      Unused    ; Serial port transmit interrupt vector
TpVect:  B      Unused    ; Software TRAP instruction vector
;*
;*****
.text
;*****
Start:
;* Main Routine for speech 'recording' using silence compression.
;*****
CALL    InitTms320      ; Set the initial TMS320 configuration
CALL    InitDataTransfer ; Initialize data transfer variables
CALL    InitMuLawExp    ; Initialize Mu-law expansion routine
CALL    InitSilenceComp ; Initialize the silence compression variables
CALL    InitAvgs        ; Initialize speech and silence interval avgs.
MainLoop:
CALL    GetSpeechFrame  ; Read the next speech frame into memory
CALL    EvaluateFrame   ; Is this frame 'speech' or 'silence'
CALL    ProcessFrame    ; Process the frame (speech or silence)
BIOZ    MainLoop        ; Get next frame, unless BIO high to quit
StopSampling:
CALL    WriteEndCode    ; Write the end of data code to memory
SXF     ; Setting external flag signals finished to PC
ZAC
SACL    IMR             ; Disable all interrupts
IDLE    ; And halt.
;*
;*****
.end
;*****

```

SilPly.asm

```

;*
;* SilPly.asm
;* May 17, 1990
;* Chris Rose
;*
;* This file contains the main routine for the silence inserted Mu-law
;* playback program.
;*****
;* Include defines file
.include "silrec.inc"
;*
;* External references
.ref    InitTms320, InitDataTransfer
.ref    Unused, Transmit, DataTransmit, CheckSilCode, CheckOverflow
.ref    GetNoise, Mult, NoiseVal, InitNoise
.ref    MuLawComp, InitMuLawComp, MuLawVal
.def    LinVal, SilAvg
;*
LinVal    .usect "B2", 1
SilAvg    .usect "B2", 1
;*****

```

```

.sect "IRUPTS"
;*****
;* Interrupt Vectors
RsVect:  B      Start    ; Reset vector
I0Vect:  B      Unused    ; Hardware interrupt 0 vector
I1Vect:  B      Unused    ; Hardware interrupt 1 vector
I2Vect:  B      Unused    ; Hardware interrupt 2 vector
.space 100h                ; reserved interrupt vector space
TmVect:  B      Unused    ; Internal timer interrupt vector
RxVect:  B      Unused    ; Serial port receive interrupt vector
TxVect:  B      Transmit  ; Serial port transmit interrupt vector
TpVect:  B      Unused    ; Software TRAP instruction vector
;*
;*****
.text
;*****
Start:
;*****
CALL    InitTms320      ; Set TMS320 & serial port configuration
CALL    InitDataTransfer ; Initialize data transfer variables
CALL    InitNoise
CALL    InitMuLawComp
MainLoop:
CALL    CheckSilCode    ; Check if silence code is next.
BNC     MainNotCode     ; Branch to play if it is not a code
LARP    CDP             ; Make data pointer active
LAC     *               ; Load the silence interval length
SBLK    EndCodeWord     ; Check for end-code
BZ      MainEnd         ; Branch if 'end code' is present
LAR     SIL,++          ; Load the silence interval length (in frames)
CALL    CheckOverflow

LAC     ++
SACL    Mult
CALL    CheckOverflow

LARP    SIL
MAR     *-              ; Adjust loop count in SIL register
SilenceDelay:
LARK    AR7, (FrameSize-1) ; Load the interval count for each frame
InnerDelay:
CALL    GetNoise
LAC     NoiseVal
SACL    LinVal
CALL    MuLawComp
LAC     MuLawVal
SACL    DXR

IDLE    ; Wait for silence to transmit
LARP    AR7
BANZ    InnerDelay, *-    ; Branch until frame is complete
LARP    SIL
BANZ    SilenceDelay, *- ; Branch for all frames of silence
B       MainLoop         ; Check for second silence code
MainNotCode:
CALL    DataTransmit     ; Move data to transmit register and transmit
B       MainLoop        ; Branch for next data
MainEnd:
LALK    00FFh
SACL    DXR             ; 'Zero' the transmit data register

```



```

;* Assumes:
;* DP = 0      Data page containing the local variables
;* ?SXM       Sign extension can be either on or off
;*
;* Passed:
;* FmEnergy    The frame energy average
;* FmZeroCross The frame zero crossings
;*
;* Returns:
;* FrameType   A flag stating the type of the current frame (0 - Sil, 1 - Sp)
;*****
ZALS  Threshold      ; Load the current energy threshold ...
SUBS  FmEnergy       ; and compare with current frame energy
BLEZ  EFCurSpeech    ; Branch if (FmEnergy >= Threshold)
LACK  ZCThresh       ; Load the zero-crossing threshold ...
SUBS  FmZeroCross    ; and compare with frame zero-crossings
BLEZ  EFCurSpeech    ; Branch if (FmZeroCross >= Threshold)
EFCurSpeech:
ZAC                      ; Current frame is 'silence' ...
SACL  FrameType       ; so store 0 as FrameType
RET
EFCurSpeech:
LACK  1                ; Current frame is 'speech' ...
SACL  FrameType       ; so store 1 as FrameType
RET
;*****
ProcessFrame:
;* Routine to process a frame that has been previously sampled, based on the
;* current FrameType, and the current silence/speech state (SpeechState).
;*
;* Assumes:
;* DP = 0      Data page containing the local variables
;* ?SXM       Sign extension can be either on or off
;*
;* Passed:
;* FrameType   A flag stating the type of the current frame (0 - Sil, 1 - Sp)
;* SpeechState A flag stating the current state (0 - Sil, 1 - Sp)
;*
;* Returned:
;* SpeechState The speech state flag is modified based on the current frame
;*
;* Modified:
;* ARP         Used to point to the various auxiliary registers
;*****
LAC  SpeechState      ; Load the current State to accum.
BNZ  PFSilSpeech      ; Branch if State=Speech (1)
PFSilSpeech:
LAC  FrameType        ; Load the current frame type to accum.
BNZ  PFSilSpeech      ; Branch if Frame=Speech (1)
B    PFSilSil         ; Otherwise go to handle Silence frame (0)
PFSilSpeechState:
LAC  FrameType        ; Load the current frame type to accum.
BNZ  PFSilSpeech      ; Branch if Frame=Speech (1)
B    PFSilSpeechSil   ; Otherwise go to handle Silence frame (0)
;*****
PFSilSil:
; State = Silence, Current Frame = Silence
CALL  UpdateSilAvg    ; Add silence frame energy to silence average
SAR   SIL,TempData    ; Store SIL to a memory location ...

```

```

ZALS  TempData        ; and load it to the accumulator, no ext.
SAR   SPC,TempData    ; Store current speech length ...
ADDS  TempData        ; and add to SIL in accumulator, no ext.
ADDS  1               ; Increment SIL for current frame
SACL  TempData        ; Store final value for SIL
SUBS  MaxSilVar       ; Compare with max allowed value, no ext.
BGEZ  PFSilMaxSil     ; Check if (SIL >= MaxSil) ...
LAR   SIL,TempData    ; SIL ok, so save it
B     PFSilSilOK
PFSilMaxSil:
LRLK  SIL,MaxSil      ; SIL too large, save max allowed
PFSilSilOK:
LARK  SPC,0           ; Set current speech length to 0
CALL  ResetCDP        ; Reset the CDP to the start of the silence
CALL  WriteSilCode     ; Write the silence code and interval length
LARP  SIL             ; Make silence length counter active again
LARK  AR0,LTHangOver   ; Load the low threshold hangover amount
CMPR  0               ; Check if (SIL = LTHangOver) ...
BBZ   PFSilHangOverOK ; branch if not
LALK  HighThresh      ; Load the high energy threshold ...
SACL  ThreshMult      ; and save as the current threshold
PFSilHangOverOK:
CALL  UpdateThreshold ; Update the energy threshold
RET
;*****
PFSilSpeech:
; State = Silence, Current Frame = Speech
LARP  SPC             ; Make speech length counter active
MAR   **              ; Increment the number of speech intervals
LALK  LowThresh       ; Load the low energy threshold ...
SACL  ThreshMult      ; and save as the current threshold
LARK  AR0,MinSpeech   ; Load the minimum stored speech allowed
CMPR  0               ; Test if (SPC = MinSpeech) ...
BBZ   PFSilSpStateOK  ; and branch if not
CALL  UpdateSpeechAvg ; Add speech frame energy to speech average
LACK  1               ; Load 1 to the accumulator
SACL  SpeechState     ; and store to signal SpeechState (1)
LARK  SIL,0           ; Update current silence length to zero
CALL  UpdateNDP       ; Update NDP to current location, o/p block #
PFSilSpStateOK:
CALL  UpdateThreshold ; Update the energy threshold
RET
;*****
PFSilSpeechSil:
; State = Speech, Current Frame = Silence
LARP  SIL             ; Make Silence Length counter active
MAR   **              ; Increment the number of silence intervals
LARK  AR0,LTHangOver  ; Load the low energy thresh hangover amount
CMPR  0               ; Test if (SIL = LTHangOver) ...
BBZ   PFSilSpHangOverOK ; and branch if not.
LALK  HighThresh      ; Load the high energy threshold ...
SACL  ThreshMult      ; and save as the current threshold
PFSilSpHangOverOK:
LARK  AR0,MinSil      ; Load the minimum stored silence allowed
CMPR  0               ; Test if (SIL = MinSil) ...
BBZ   PFSilSpStateOK  ; and branch if not.
CALL  UpdateSilAvg    ; Add silence frame energy to silence average
ZAC                      ; Zero the accumulator ...
SACL  SpeechState     ; and store to signal SilenceState (0)
LARK  SPC,0           ; Update current speech length to zero
CALL  ResetCDP        ; Reset the CDP to start of silence
CALL  WriteSilCode     ; Write the silence code and silence length

```



```

SAR    AVG,SPCBLKPnt
LRLK   AVG, (SpeechAvgs+SPCBLKLEN) ; Evaluate end of SpeechAvgs array
SAR    AVG,ESpeechAvgs
LRLK   AVG, (SpeechBlks+SPCNUMBLK) ; Evaluate end of SpeechBlks array
SAR    AVG,ESpeechBlks

;* Now, initialize silence average variables
LDPK   SILPAGE ; Switch to data page with silence variables

;* Initialize the silence interval average and associated array
LARP   AVG ; Use AVG register as a pointer
LRLK   AVG,SilAvgs ; Point to first entry in SilAvgs array
LALK   INITSILAVG ; Load the initial value for SilAvg
RPTK   (SILAVGLEN-1) ; Fill complete SilAvgs array ...
SACL   ** ; with the initial value for SilAvg
SACL   SilAvg ; Save initial SilAvg

;* Initialize the variable used in calculating the array sum
ZALH   INITSILAVG ; Load InitSilAvg*SilAvgLen to accum.
SACL   LSilAvgSum ; Save in SilAvgSum
SACH   HSilAvgSum

;* Initialize the pointers for the silence average array
LRLK   AVG,SilAvgs ; Initialize SilAvgs pointer
SAR    AVG,SilAvgPnt
LRLK   AVG, (SilAvgs+SILAVGLEN) ; Evaluate end of SilAvgs array
SAR    AVG,ESilAvgs

;* Initialize the "constant" variables
LALK   ECRIT ; Initialize the ECRIT variable
SACL   ECRIT
LALK   CSPSCALE ; Initialize the SpeechScale variable
SACL   SpeechScale

LDPK   REGPAGE ; Switch back to default data page
RET

;*
*****
UpdateSilAvg:
;* This routine updates the silence interval average. The current frame is
;* known to be silence, and that frames energy is added to the average
;* calculation.
;* Note: Sign extension must be set
*****
;* First, copy the frame energy into local data space
LAC    FmEnergy ; Load the current frame energy (page 0)
LDPK   SILPAGE ; Change to page containing silence variables
SACL   SilEnergy ; Store frame energy in local space

;* Check if the SilAvg array needs to be initialized. This is indicated by
;* a negative number in SilAvg (a flag).
LAC    SilAvg ; Initialize SilAvgs array if current ...
BLZ    InitializeSilAvg ; SilAvg < 0

;* Include this value of frame energy only if ENERGY < ECRIT * SILAVG. This
;* prevents SilAvg from being distorted by unusually large frame energies.
LT     SilAvg ; Prepare to multiply by SilAvg (Q0)
MPY    ECRIT ; Multiply by ECRIT (Q12)

PAC    ; Place product in accumulator (Q12)
SACH   TempData,4 ; Save result for shifting (Q0)
LAC    TempData ; Reload integer product
SUB    SilEnergy ; Check if SilEnergy <= ECRIT*SilAvg ...
BLZ    EndUpdateSil ; if not, do not include this energy

;* Update the silence interval average and the associated array using the
;* current frame energy.
LARP   AVG ; Use AVG register as a pointer
LAR    AVG,SilAvgPnt ; Point to current location in SilAvgs array
ZALS   LSilAvgSum ; Load the current array sum to the accum.
ADDDH   HSilAvgSum
SUB     *,(16-SILAVGSFT) ; Subtract old frame energy from current loc.
ADD     SilEnergy,(16-SILAVGSFT) ; Add new frame energy to sum
SACH   SilAvg ; Save new silence energy average (high acc.)
SACL   LSilAvgSum ; Save new sum in SilAvgSum
SACH   HSilAvgSum
LAC    SilEnergy ; Reload new frame energy ...
SACL   ** ; and save to current array location
SAR    AVG,SilAvgPnt ; Save updated array pointer

;* Check for wrap-around in the silence average array pointer
LAR    ARO,ESilAvgs ; Load the end-of-array pointer value to ARO
CMR    1 ; TC=1 if (ESilAvgs > AVG) ie, not at end yet
BRNZ   SilAvgPntOk ; Branch if TC=1
LRLK   AVG,SilAvgs ; Wrap-around existed, so reset the array ...
SAR    AVG,SilAvgPnt ; pointer to the start of the array
SilAvgPntOk:
B       EndUpdateSil ; Update complete so jump to exit

;* Initialize SilAvg and the associated array to the value of the silence
;* interval frame energy.
InitializeSilAvg:

;* Initialize the silence interval average and the associated array
LARP   AVG ; Use AVG register as a pointer
LRLK   AVG,SilAvgs ; Point to first entry in SilAvgs array
LAC    SilEnergy ; Load the initial value for SilAvg
RPTK   (SILAVGLEN-1) ; Fill complete SilAvgs array ...
SACL   ** ; with the initial value for SilAvg
SACL   SilAvg ; Save initial SilAvg

;* Initialize the variable used in calculating the array sum
ZALH   SilEnergy ; Load SilEnergy*SilAvgLen to accum.
SACL   LSilAvgSum ; Save in SilAvgSum
SACH   HSilAvgSum

;* Initialize the pointers for the silence average array
LRLK   AVG,SilAvgs ; Initialize SilAvgs pointer
SAR    AVG,SilAvgPnt
LRLK   AVG, (SilAvgs+SILAVGLEN) ; Evaluate end of SilAvgs array
SAR    AVG,ESilAvgs

EndUpdateSil:
LDPK   REGPAGE ; Switch back to default data page
RET
;*
*****

```

```

UpdateSpeechAvg:
;* This routine updates the speech interval average energy. The current frame
;* is known to be speech, and that frames energy is added to the average
;* calculation.
;* Note: Sign extension must be set
;*****
;* First, copy the frame energy into local data space
    LAC    FmEnergy      ; Load the current frame energy (page 0)
    LDPK   SPEECHPAGE    ; Change to page containing speech variables
    SACL   SpcEnergy     ; Save the frame energy in local data space

;* Check if the SpeechAvg arrays need to be initialized. This is indicated by
;* a negative number in SpeechAvg (a flag).
    LAC    SpeechAvg     ; Initialize SpeechAvgs and SpeechBlks ...
    BLZ    InitializeSpcAvg ; arrays if current SpeechAvg < 0

;* Update the speech interval energy average and the associated arrays using
;* the current frame energy.

;* First, update the current speech interval block average
    LARP   AVG           ; Use the AVG register as a pointer
    LAR    AVG, SpcAvgPnt ; Point to the current loc. in block avg array
    ZALS   LSpcAvgSum     ; Load the current block sum to accum.
    ADDH   HSpcAvgSum
    ADD    SpcEnergy, (16-SPCAVGSFT) ; Add current energy to sum
    SACH   SpcBlkAvg      ; Save current block average (high accum)
    SACL   LSpcAvgSum     ; Save the updated sum to SpcAvgSum
    SACH   HSpcAvgSum
    LAC    SpcEnergy     ; Reload the frame energy to the accum ...
    SACL   ++            ; and save to current array location
    SAR    AVG, SpcAvgPnt ; Save the updated speech block array pointer

;* Check if the current speech block average is complete (end of block)
    LAR    AR0, ESpeechAvgs ; Load the end of array pointer value to AR0
    CMPR   1               ; TC=1 if (ESpeechAvgs > AVG) ie, not at end
    BBNZ   SpcAvgPntOk     ; Branch if TC=1

;* Current block is full, so reset array pointer and array sum
    ZAC    ; Zero the accumulator ...
    SACL   LSpcAvgSum     ; and save to SpcAvgSum
    SACH   HSpcAvgSum
    LALK   SpeechAvgs    ; Reset array pointer to start of the array
    SACL   SpcAvgPnt

;* Now, add the block average to the speech interval energy average
    LARP   AVG           ; Use the AVG register as a pointer
    LAR    AVG, SpcBlkPnt ; Point to the current loc. in blocks array
    ZALS   LSpcBlkSum     ; Load the current array sum to the accum.
    ADDH   HSpcBlkSum
    SUB    *, (16-SPCBLKSFT) ; Subtract the old block average ...
    ADD    SpcBlkAvg, (16-SPCBLKSFT) ; and add the new one
    SACH   SpeechAvg      ; Save the new speech average (high accum)
    SACL   LSpcBlkSum     ; Save the updated array sum
    SACH   HSpcBlkSum
    LAC    SpcBlkAvg      ; Reload the block average ...
    SACL   ++            ; and save to the current array location
    SAR    AVG, SpcBlkPnt ; Save the updated array pointer

;* Check for wrap-around in the array pointer
    LAR    AR0, ESpeechBlks ; Load the pointer value for array end to AR0

```

```

    CMPR   1               ; TC=1 if (ESpeechBlks > AVG) ie, not at end
    BBNZ   SpcBlkPntOk    ; Branch if TC=1
    LRLK   AVG, SpeechBlks ; Wrap-around existed, so reset the ...
    SAR    AVG, SpcBlkPnt ; pointer to the start of the array

SpcBlkPntOk:
SpcAvgPntOk:
    B      EndUpdateSpc    ; Average update is done, so jump to exit

;* Initialize the speech average and the associated arrays to the current
;* frame energy value.
InitializeSpcAvg:

;* Initialize the speech interval average, and associated array. The
;* SpeechAvgs need not be initialized as this array is summed from the start
;* for each new block.
    LARP   AVG           ; Use AVG register as a pointer
    LRLK   AVG, SpeechBlks ; Point to first entry in SpeechBlks array
    LAC    SpcEnergy     ; Load the initial value for SpeechAvg
    RTRK   (SPCNUMBLK-1) ; Fill complete SpeechBlks array ...
    SACL   ++            ; with the initial value for SpeechAvg
    SACL   SpeechAvg      ; Save initial SpeechAvg

;* Initialize the two variables used in calculating the array sums
    ZALH   SpcEnergy     ; Load SpcEnergy*SpCNumBlks to accum.
    SACL   LSpcBlkSum    ; Save in SpcBlkSum
    SACH   HSpcBlkSum
    ZAC    ; Zero the accumulator ...
    SACL   LSpcAvgSum    ; and save in SpcAvgSum
    SACH   HSpcAvgSum

;* Initialize the pointers for the two speech average arrays
    LRLK   AVG, SpeechAvgs ; Initialize SpeechAvgs pointer
    SAR    AVG, SpcAvgPnt
    LRLK   AVG, SpeechBlks ; Initialize SpeechBlks pointer
    SAR    AVG, SpcBlkPnt
    LRLK   AVG, (SpeechAvgs+SPCBLKLEN) ; Evaluate end of SpeechAvgs array
    SAR    AVG, ESpeechAvgs
    LRLK   AVG, (SpeechBlks+SPCNUMBLK) ; Evaluate end of SpeechBlks array
    SAR    AVG, ESpeechBlks

EndUpdateSpc:
    LDPK   REGPAGE       ; Switch back to default data page
    RET

;*****
UpdateThreshold:
;* This routine will update the actual energy threshold based on the current
;* silence interval average energy, the speech interval average energy, and
;* the current energy level multiplier.
;*****
;* First, check if the speech interval energy average and the silence interval
;* is valid (a negative number signifies the energy average has not yet been
;* initialized).
    LDPK   SPEECHPAGE    ; Change to speech variables page ...
    LAC    SpeechAvg      ; and load the speech interval energy avg.
    LDPK   SILPAGE       ; Change to silence page for later calcs.
    BLZ    UseInitScale  ; Branch if speech average is not valid

```

```

    LAC    SilAvg      ; Load the silence interval energy avg.
    BCEZ   SilAvgOk    ; Branch if the silence average is Ok.

; * Here, either the speech or silence energy average was invalid, therefore
; * use a default threshold scaler.
UseInitScale:
    LALK   INITSCALE   ; Load the default scaler ...
    SACL   Scale       ; and save to the scale variable
    B      CalcThresh  ; Branch to calculate the threshold

; * Now, calculate the threshold scaler based on the dual adaptive scheme
; * using the long term silence and speech interval energy averages.
SilAvgOk:
    LDPK   SPEECHPAGE  ; Switch to speech variables ...
    ZALS   SpeechAvg   ; to reload the speech average
    LDPK   SILPAGE     ; Switch back to silence variables
    RPTK   15          ; Do next instruction 16 times ...
    SUBC   SpeechScale ; to divide SpeechAvg by SpeechScale
    SACL   TempData     ; Store the temporary result
    LAC    SilAvg       ; Load the silence interval energy average ...
    SUB    TempData     ; and compare with scaled speech value
    BLZ    UseSpeech    ; Use the larger of silence or scaled speech

UseSilence:
    LAC    SilAvg       ; Silence was larger, so save SilAvg ...
    SACL   Scale        ; in the scale variable
    B      CalcThresh  ; Now calculate the actual threshold

UseSpeech:
    LAC    TempData     ; Scaled speech was larger, so save SilAvg ...
    SACL   Scale        ; in scale variable

; * Finally, calculate the actual energy threshold, based on the threshold
; * scaler calculated above, and the threshold multiplier determined by the
; * speech detection algorithm.
CalcThresh:
    LT     Scale        ; Load the scaler in preparation for mult.
    LDPK   REGPAGE     ; Switch back to default variable page
    MPY    ThreshMult   ; Multiply by threshold multiplier (Q12)
    PAC    ThreshMult   ; Load result to accumulator (Q12)
    SACH   Threshold,4  ; Save the new energy threshold (Q0)
    RET

; *
; *****
; .end
; *****

```

Noise.asm

```

; *
; * Noise.asm
; * June 26, 1990
; * Chris Rose
; *
; * This file contains the routines to implement a Gaussian noise generator.
; *****
; *
; * External References
; .def    GetNoise, Mult, NoiseVal, InitNoise
; *

```

```

; * Define internal variables used in the routine (Block B2, page 0)

```

```

HighSeed .usect "B2", 1
LowSeed  .usect "B2", 1
HighA    .usect "B2", 1
LowA     .usect "B2", 1
HighC    .usect "B2", 1
LowC     .usect "B2", 1
Mult     .usect "B2", 1
NoiseVal .usect "B2", 1
HighTemp .usect "B2", 1
LowTemp  .usect "B2", 1
RanU     .usect "B2", 1
RanG     .usect "B2", 1
Scale    .usect "B2", 1

```

```

; *

```

```

; *****

```

```

; .text

```

```

; *****

```

```

InitNoise:

```

```

; *****

```

```

    ZAC

```

```

    SACL   HighSeed

```

```

    SACL   LowSeed

```

```

    LALK   03FFh

```

```

    SACL   HighA

```

```

    LALK   944Dh

```

```

    SACL   LowA

```

```

    LALK   00D8h

```

```

    SACL   HighC

```

```

    LALK   658Bh

```

```

    SACL   LowC

```

```

    LALK   13F8h

```

```

    SACL   Scale

```

```

    RET

```

```

; *

```

```

; *****

```

```

GetNoise:

```

```

; *****

```

```

    CALL   GetGauss

```

```

    LT     Scale

```

```

    MPY    RanG          ; Scale the RanG value by 1.248 (Q24)

```

```

    PAC

```

```

    SACH   RanG,4        ; Save scaled RanG (Q12)

```

```

    LT     RanG

```

```

    MPY    Mult          ; Multiply noise by desired energy value (Q12)

```

```

    PAC

```

```

    SACH   NoiseVal,4    ; Save noise desired noise value (Q0)

```

```

    RET

```

```

; *

```

```

; *****

```

```

GetGauss:

```

```

; *****

```

```

    LALK   0A000h        ; Load -6 (Q12) to AccL

```

```

    SACL   RanG          ; And save temporary value

```

```

    LARK   AR0,11        ; Loop 12 times

```

```

    LARP   AR0

```

```

GaussLoop:

```

```

    CALL   GetUniform    ; Get a uniform [0,1] Q12 number

```

```

    ZALS   RanG          ; Load RanG running total

```

```

    ADDS   RanU          ; Add the current uniform number

```

```

SACL  RanG          ; Save updated RanG
BANZ  GaussLoop,*-
RET
;
;*****
GetUniform:
;*****
LT     LowSeed       ; T = X0
MPYU   LowA          ; T = X0, P = X0 * A0
SPL    LowTemp       ; Save W0
SPH    HighTemp      ; Save partial W1
MPYU   HighA         ; T = X0, P = X0 * A1
LTP    HighSeed      ; T = X1, P = X0 * A1, Acc = X0 * A1
MPYU   LowA          ; T = X1, P = X1 * A0, Acc = X0 * A1
APAC   ; Acc = X0*A1 + X1*A0
ADDS   HighTemp      ; Acc = X0*A1 + X1*A0 + X0*A0*2**-16
ANDK   03FFh         ; Mask off lower 10 bits to prevent overflow
SACL   HighTemp      ; Save W1
ZALH   HighTemp
ADDS   LowTemp       ; Load W1W0 to Acc
ADDS   LowC
ADDS   HighC         ; Add C to Acc
SACL   LowSeed
ANDK   3FF0h,12      ; Mask off 10 low bits in AccH
SACH   HighSeed      ; And save new Seed
ADDS   LowSeed       ; Restore new seed to accum.
SACH   RanU,2        ; Store Uniform random number in Q12 format
RET
;
;*****
;end
;*****

```

uLawExp.asm

```

;
; * uLawExp.asm
; * May 16, 1990
; * Chris Rose
;
; * This file contains the routine to convert 8-bit Mu-law compressed PCM data
; * to 14-bit linear PCM data.
;*****
; *
; * External references
; * .def MuLawExp, InitMuLawExp
; * .ref LinVal, MuLawVal
; *
; * Define internal variables used in the routine (Block B2, page 0)
Bias .usect "B2", 1 ; Bias of 33 used in conversion
LinSeg .usect "B2", 1 ; Segment of linearization
Sum .usect "B2", 1 ; Intermediate storage for conversion
;
;*****
; .text
;*****
InitMuLawExp:
; * This routine initializes the constant used in the Mu-law expansion

```

```

; * routine (MuLawExp). This routine must be called before the first Mu-law
; * conversion.
; *
; * Assumes:
; * DP = 0 Data page containing the local variables
; * ?SXM Sign extension can be either on or off
;*****
LACK 33 ; Load the Bias constant
SACL Bias ; and store it
RET
;
;*****
MuLawExp:
; * This routine performs the actual Mu-law PCM to linear PCM conversion.
; *
; * Assumes:
; * DP = 0 Data page containing the local variables
; * ?SXM Sign extension can be either on or off
; *
; * Passed:
; * MuLawVal 8-bit Mu-law value to be converted to a linear value
; *
; * Returned:
; * LinVal 14-bit linear value, right justified in a 16-bit word
; *
; * Modified:
; * AR0 Used to point to LinVal & MuLawVal variables
; * ARP Used to point to AR0
;*****
LRLK AR0,MuLawVal ; Load AR0 with address of MuLawVal
LARP AR0 ; use AR0 for indirect addressing (MuLawVal)
BIT *,8 ; store sign bit of MuLawVal in TC for later use
LAC *,12 ; load Rx byte into Acc, straddle high & low nib
ANDK 7F00h,4 ; zero sign and other unused bits in acc.
XORK 7F00h,4 ; invert Rx data bits
SACH LinSeg ; store high acc., the segment
SUBH LinSeg ; zero the high acc.
ADD Bias,11 ; add bias, offset one bit right of Rx low nib.
SFL ; shift acc right 1 bit (gives 2Q+33)
SACH Sum,4 ; store 2Q+33 (low 4 bits were in low acc)
LT LinSeg ; load T reg with segment = shift value
LACT Sum ; re-load acc, with shift specified in T reg
SUB Bias ; subtract off the bias
BBNZ PosVal ; branch if original number positive (TC=1)
NEG ; 2's complement negate to make negative
PosVal:
LRLK AR0,LinVal ; point to LinVal
SACL *,0,AR0 ; store converted value to LinVal variable
RET ; and return
;
;*****
;end
;*****

```

uLawComp.asm

```

;*
;* uLawComp.asm
;* May 16, 1990
;* Chris Rose
;*
;* This file contains the routine to convert 14-bit linear PCM data to 8-bit
;* Mu-law compressed data.
;*****
;* External references
; .def      MuLawComp, InitMuLawComp
; .ref      LinVal, MuLawVal
;*
;* Define internal variables used in the routine (Block B2, page 0)
Bias2      .usect "B2", 1      ; Bias of 33 left shifted 2 bits
Neg7       .usect "B2", 1      ; Storage for -7 constant
LinSeg     .usect "B2", 1      ; Segment of linearization
Quant      .usect "B2", 1      ; Quantization bin
;*****
InitMuLawComp:
;* This routine initializes the constants used in the Mu-law compression
;* routine (MuLawComp). This routine must be called before the first Mu-law
;* conversion.
;*
;* Assumes:
;* DP = 0      Data page containing the local variables
;* ?SXM       Sign extension can be either on or off
;*****
LACK      132      ; load bias constant, 33 left shifted 2 bits
SACL      Bias2      ; and store to bias variable
ZAC
SUBK      7      ; Load -7 to the accumulator
SACL      Neg7      ; and store it
RET      ; and return to calling routine
;*
;*****
MuLawComp:
;* This routine performs the actual linear PCM to Mu-law PCM conversion.
;*
;* Assumes:
;* DP = 0      Data page containing the local variables
;* SXM       Sign extension mode must be on for -ve 16-bit values
;*
;* Passed:
;* LinVal      14-bit, right justified, sign-extended linear value
;*
;* Returned:
;* MuLawVal     8-bit Mu-law representation of LinVal
;*
;* Modified:
;* AR0       Used to normalize exponents, and point to LinVal & MuLawVal
;* ARP       Used to point to AR0
;*****
LRLK      AR0, LinVal      ; make AR0 point to LinVal for addressing
LARP      AR0
LAC      *,2      ; Left justify linear value into low accum.

```

```

SACL      Quant      ; Store left justified value to temp. location
ZALH      Quant      ; and restore in high accumulator
BLZ       NegCompute  ; Test for negative number (Separate calc.)
;*
;* Here, a positive value is being converted
;*
ADDH      Bias2      ; add bias to left justified number
LAR        AR0, Neg7    ; load -7 to AR0 (already pointed to)
RPTK      6          ; repeat NORM 7 times
NORM      ++         ; Shift 1 into bit 30 of acc. store shift in AR0
ANDK      0F000h, 14   ; zero all bit 4 Q bits (2 MSB's and all LSB's)
SACH      Quant      ; and store Q bits
SAR        AR0, LinSeg  ; store negative segment count
ZALH      LinSeg      ; and restore to high accumulator
ABS        ; convert to positive value
ADD        Quant, 2     ; add Q bits, left shifted 2 to MSB's of low acc
XORK      0FF00h, 4     ; invert all 8 bits to transmit
B          Compressed   ; and jump to store converted word
NegCompute:
;*
;* Here a negative value is being converted
;*
ABS        ; negative number; convert to positive
ADDH      Bias2      ; add bias to left justified number
LAR        AR0, Neg7    ; load -7 to AR0 (already pointed to)
RPTK      6          ; repeat NORM 7 times
NORM      ++         ; Shift 1 into bit 30 of acc. store shift in AR0
ANDK      0F000h, 14   ; zero all bit 4 Q bits (2 MSB's and all LSB's)
SACH      Quant      ; and store Q bits
SAR        AR0, LinSeg  ; store negative segment count
ZALH      LinSeg      ; and restore to high accumulator
ABS        ; convert to positive value
ADD        Quant, 2     ; add Q bits, left shifted 2 to MSB's of low acc
XORK      7F00h, 4     ; invert 7 magnitude bits; sign bit = 0
Compressed:
;*
;* Store the converted number to the MuLawVal variable
;*
LRLK      AR0, MuLawVal  ; point AR0 to MuLawVal
SACH      *,4          ; write converted number to MuLawVal storage
RET
;*****
.end
;*****

```

RxIntrpt.asm

```

;*
;* RxIntrpt.asm
;* May 1, 1990
;* Chris Rose
;*
;* This file contains the interrupt handling routines to receive Mu-law data
;* from the serial port, and 'pack' it into words in external data memory.
;*****
;*
;* Include defines file

```

```

.include "silrec.inc"
;*
;* External references
.def Receive, Unused, SamplePos, MuLawVal, LinVal
.ref CheckOverflow, MuLawExp
;*
;* Define external data variables (Block B2)
SamplePos .usect "B2", 1 ; Signals position in word to save next sample
MuLawVal .usect "B2", 1 ; MuLaw version of the current sample
LinVal .usect "B2", 1 ; Linear version of the current sample
;*
*****
.text
*****
Receive:
;* Serial port receive interrupt handler.
;* This routine receives data from the serial port, packs it into words,
;* and stores it in external memory.
;*
;* Assumes:
;* DP = 0 Data page containing the local variables
;* ?SXM Sign extension can be either on or off
;*
;* Returns:
;* MuLawVal 8-bit received Mu-law data sample
;* LinVal 14-bit (right just) version of the received value
;*
;* Modified:
;* ARP Used to point to CDP and TMP auxiliary registers
*****
;* Load the received 8-bit data
;*
LAC DRR ; Load the received data
ANDK 00FFh ; Mask off the low 8 bits
SACL MuLawVal ; And save the Mu-law value
CALL MuLawExp ; Convert Mu-law data to linear in LinVal
;*
;* Now save the 8-bit data to external memory
;*
LARP TMP ; Make TMP register active
LAR TMP, SamplePos ; Load H/L byte flag (1/0) to TMP
BANZ RxHighByte, ++, CDP ; Branch if Flag=1; Make data pointer active
RxLowByte:
SAR TMP, SamplePos ; Store the flag = 1 (Next sample high byte)
LAC MuLawVal ; Load the received data into the accumulator
SACL * ; Store the low byte at the current location
B RxEndSave
RxHighByte:
LARK TMP, 0 ; Set TMP register to 0 so will reset flag
SAR TMP, SamplePos ; Store the flag = 0 (Next sample low byte)
LAC MuLawVal, 8 ; Load data to high bits of low accum.
OR * ; Add low bits from previous sample
SACL * ; Store data word at current data location
LAC ** ; Reload the packed data word
CALL CheckOverflow
SBLK SilCodeWord ; Check if current word = code word
BNZ RxEndSave ; Branch if not equal
SACL ** ; Store 0000h in next location
CALL CheckOverflow

```

RxEndSave:

```

RET
;*
*****
Unused:
;* Default interrupt handler for unused handlers (should not get called)
*****
EINT ; simply re-enable interrupts,
RET ; and return to previous task.
;*
*****
.end
*****

```

TxIntrpt.asm

```

;*
;* TxIntrpt.asm
;* May 15, 1990
;* Chris Rose
;*
;* This file contains the routine to extract the Mu-law data from external
;* data memory, and transmit it through the serial port. This file also
;* contains the interrupt handlers.
*****
;*
;* Include defines file
.include "silrec.inc"
;*
;* External references
.def Transmit, Unused, DataTransmit, SamplePos, MuLawVal
.ref CheckOverflow, UpdateNDP
;*
;* Define data variables
SamplePos .usect "B2", 1 ; Signals position in word to save next sample
MuLawVal .usect "B2", 1 ; 8-bit Mu-law version of the current sample
;*
*****
.text
*****
DataTransmit:
;* This routine extracts Mu-law PCM data from the current data location, and
;* transmits it through the serial port.
;*
;* Assumes:
;* DP = 0 Data page containing the local variables
;* ?SXM Sign extension can be either on or off
;*
;* Returns:
;* MuLawVal 8-bit Mu-law version of current data sample
;*
;* Modified:
;* ARP Used to point to CDP and TMP auxiliary registers
*****
;*
;* Get 8-bit Mu-law data from external memory
;*
LARP TMP ; Make TMP register active

```



```

LAR    TMP,SamplePos    ; Load H/L byte flag (1/0) to TMP
BANZ   TxHighByte,++,CDP ; Branch if Flag=1; Make data pointer active
TxLowByte:
SAR     TMP,SamplePos    ; Store the flag = 1 (Next sample high byte)
LAC     *                ; Load new data value, do not alter pointer
ANDK    00FFh           ; Mask off low 8 bits
SACL    MuLawVal         ; Store low 8 bits to MuLawVal
B       TxEndGet
TxHighByte:
LARR    TMP,0           ; Set TMP register to 0 so will reset flag
SAR     TMP,SamplePos    ; Store the flag = 0 (Next sample low byte)
LAC     **,,8           ; Get high byte (inc point) in high accum
ANDK    0FF0h,12        ; Mask off low 8 bits of high accumulator
SACH    MuLawVal         ; Store high 8 bits to MuLawVal
CALL    CheckOverflow
TxEndGet:
;*
;* Now transmit the data
;*
LAC     MuLawVal         ; Load the current MuLaw sample
SACL    DXR              ; Save sample to transmit register
IDLE    ; Wait for interrupt (ie, data to be transmit)
CALL    UpdateNDP        ; Update NDP to current position, o/p block #
RET

;*****
Transmit:
;* Actual transmit interrupt handler.
;* This is called after the serial port has transmitted the previous data
;* so simply return.
;*
;* Assumes:
;* Nothing
;*****
RET
;*
;*****
Unused:
;* Default interrupt handler for unused interrupts (should not get called)
;*
;* Assumes:
;* Nothing
;*****
EINT    ; Simply re-enable interrupts
RET     ; and return
;*
;*****
.end
;*****

```

Utility.asm

```

;*
;* Utility.asm
;* May 16, 1990
;* Chris Rose
;*
;* This file contains general purpose routines to maintain data transfer

```

```

;* to/from the PC host, to initialize the Tms320, and to handle the silence
;* interval codes.
;*****
;* Include defines file
;* .include "silrec.inc"
;*
;* External references
;* .def WriteSilCode, WriteEndCode, OutputBlockNum, CheckOverflow
;* .def CheckSilCode, InitTms320, InitDataTransfer, UpdateNDP, ResetCDP
;* .ref SamplePos, SilAvg
;*
;* Define internal variables used by these procedures (Block B2, page 0)
TempData .usect "B2", 1 ; General scratch variable
DataMemSrt .usect "B2", 1 ; Offset of start of data memory
DataMemEnd .usect "B2", 1 ; Last word of external data memory
;*
;*****
.text
;*****
WriteSilCode:
;* This routine will write a silence interval code to the output data memory.
;* It writes the silence code word, followed by the silence interval length.
;*
;* Assumes:
;* DP = 0 Data page containing the local variables
;* ?SKM Sign extension can be either on or off
;*
;* Modified:
;* ARP Used to point to the CDP indirect addressing register
;*****
LARP    CDP              ; Make the current data pointer active
LALK    SilCodeWord      ; Load the silence codeword ...
SACL    **               ; and store to memory
CALL    CheckOverflow     ; Check for end of data memory
SAR     SIL,**           ; Write the interval length to memory
CALL    CheckOverflow     ; Check for end of data memory
LDPK    SILPAGE
LAC     SilAvg
LDPK    REGPAGE
SACL    **
CALL    CheckOverflow
RET
;*
;*****
WriteEndCode:
;* This routine will write the 'End of Data' code to the output data memory.
;* It writes the silence code, followed by the 'File End' code, and then
;* updates the NDP and outputs the final block number.
;*
;* Assumes:
;* DP = 0 Data page containing the local variables
;* ?SKM Sign extension can be either on or off
;*
;* Modified:
;* ARP Used to point to the CDP indirect addressing register
;*****
LARP    CDP              ; Make the current data pointer active
LALK    SilCodeWord      ; Load the silence codeword ...
SACL    **               ; and store to memory

```

```

CALL    CheckOverflow    ; Check for end of data memory
LALK    EndCodeWord      ; Load the end of data code ...
SACL    *                ; and store to memory; don't inc mem loc.
SAR     CDP,TempData     ; Save the current data pointer ...
LAR     NDP,TempData     ; to update the new data pointer
CALL    OutputBlockNum   ; Output the final block number to the PC
RET

;*
*****
OutputBlockNum:
;* Routine to output to the PC, through the CommLatch, the current block
;* number being used. The current block number is based on the New Data
;* Pointer (NDP). This routine is 'hard-coded' for 4 data blocks, each of
;* which is 2048 bytes long.
;*
;* Assumes:
;* DP = 0      Data page containing the local variables
;* ?SXM       Sign extension can be either on or off
*****
SAR     NDP,TempData     ; Store new data pointer to xfer to accum.
LAC     TempData,6       ; Load data loc. so block num in high accum.
ANDK    0C00h,6         ; Mask off only block number bits in accum.
SACH    TempData        ; Save current block number ...
OUT     TempData,CommLatch ; for output to PC
RET

;*
*****
CheckOverflow:
;* Routine to check if the data pointer has gone passed the end of data
;* memory. If it has, it is reset to the start of data memory.
;*
;* Assumes:
;* DP = 0      Data page containing the local variables
;* ARP -> CDP  The CDP register must be active (remains active)
;* ?SXM       Sign extension can be either on or off
;*
;* Modified:
;* TMP        The TMP register is used and altered in this routine
*****
LAR     TMP,DataMemEnd   ; Load last data address to AR0
CMPR    2             ; Check if CDP > DataMemEnd (overflow)
BBZ     CheckOverMemOk  ; Branch if pointer still within memory
LAR     CDP,DataMemSrt   ; Reset pointer to start of memory
CheckOverMemOk:
RET

;*
*****
CheckSilCode:
;* Routine to check if a silence code is in the data memory, starting at
;* the current data location. If a silence code is found, CDP is left
;* pointing at the 'run-length' code, and the carry flag is set; otherwise
;* CDP is left pointing to the next data word, and the carry flag is reset.
;*
;* Assumes:
;* DP = 0      Data page containing local variables
;* ?SXM       Sign extension can be on or off
;*
;* Returns:
;* Carry      Flag set if a code is found, otherwise the flag is reset

```

```

;* Modified:
;* ARP        Used to point to CDP indirect addressing register
*****
SAR     CDP,TempData     ; Store CDP in case needs restoring
LAR     CDP             ; Make data pointer active
LAC     *               ; Load next data word,
SBLK    SilCodeWord     ; Compare with code-word
BNZ     CheckNotCode    ; Branch if not a codeword
CALL    CheckOverflow
LAC     *               ; Load run-length code
BZ      CodeInData       ; Not a codeword if = 0000h
SC      ; Set carry flag, it is a code
RET     ; Return with CDP pointing to silence length
CodeInData:
LALK    SilCodeWord     ; Load the SilCode data (was followed by 0000)
SACL    *               ; Overwrite 0000 with the original data
RC      ; Reset carry flag to signal 'Not a Code'
RET

CheckNotCode:
LAR     CDP,TempData     ; Restore data pointer (it is data)
RC      ; Reset carry flag to signal 'Not a Code'
RET

;*
*****
UpdateNDP:
;* This routine will update the new data pointer (NDP) to the current data
;* location (CDP), and then output the new current block number to the PC.
;*
;* Assumes:
;* DP = 0      Data page containing local variables
;* ?SXM       Sign extension can be either on or off
*****
SAR     CDP,TempData
LAR     NDP,TempData
CALL    OutputBlockNum
RET

;*
*****
ResetCDP:
;* This routine will reset the current data location (CDP) back to the start
;* of the new data (NDP) in preparation to overwrite un-needed data.
;*
;* Assumes:
;* DP = 0      Data page containing local variables
;* ?SXM       Sign extension can be either on or off
*****
SAR     NDP,TempData
LAR     CDP,TempData
RET

;*
*****
InitTms320:
;* Routine to set the configuration of the TMS processor, and serial port
;* by setting the appropriate status bits.
;*
;* Assumes:
;* Nothing
*****
;* Set configuration for the serial port

```

```

;*
  FORT      1          ; Configure serial port for 8-bit data
  SFSM                      ; Require frame synch pulse for each byte
  RTXM                      ; Start serial transit on external FSX pulse
  LALK      00FFh
  SACL      DXR          ; 'zero' the serial port transmit register
  ZAC                      ; zero the accumulator
  SACL      DRR          ; and the serial port receive register
;*
;* Set TMS processor configuration
;*
  CNFD                      ; configure mem-block B0 as data
  SOVM                      ; Set overflow mode
  SSXM                      ; Turn on sign extension for -ve 16 bit values
  SPM        0          ; Set P-register output shift to 'no-shift'
  SHM                      ; Set hold mode (Hold halts entire TMS)
  LDPK        0          ; Use data page 0 for registers and B2 data
  RXP                      ; Reset XP (Setting XP signals finished)
;*
;* Set Interrupt mask
;*
  LALK      IntrptMask      ; load mask into accumulator
  SACL      IMR            ; and store to the mask register
  RET
;*
*****
InitDataTransfer:
;* Routine to initialize the constants and variable used in transferring
;* data between the PC host and the TMS320.
;*
;* Assumes:
;* DP = 0      Data page containing local variables
;* ?SXM        Sign extension can be either on or off
*****
  LALK      DataOffset      ; Load data offset to accumulator ...
  SACL      DataMemSrt      ; and initialize data memory start constant
  ADLK      (DataMemSize - 1) ; Find the last available data word ...
  SACL      DataMemEnd      ; and initialize data memory end constant
  LAR       CDP, DataMemSrt  ; Set data pointer to start of data memory
  LAR       NDP, DataMemSrt  ; Set new data pointer to start of memory
  CALL      OutputBlockNum   ; Output the initial block number to the PC
  ZAC
  SACL      SamplePos        ; Set to 0, next sample is first in new word
  RET
;*
*****
.end
*****

```

This file contains the routines to implement conversion between linear speech samples, and 4-bit ADPCM coded speech samples.

AdpcmCom.asm

```

;
;*
;* AdpcmCom.asm
;* May 3, 1990
;* Chris Rose
;*
;* Routine to perform 32Kbps ADPCM encoding. The algorithm used is modelled
;* after the CCITT standard, but is not identically bit compatible. The
;* algorithm is taken from "32-kbit/s ADPCM with the TMS32010", an application
;* note from Texas Instruments. The original source code was modified slightly
;* to be incorporated into the silence compression program.
;*
;* Number formats:
;*   nb      - number is stored in n bits
;*   Qn      - there are n decimal bits
;*   TC      - two's compliment notation
;*   POS     - unsigned notation
;*****
;* External references
;*   .ref    LinVal, AdpcmVal
;*   .def    AdpcmComp, ResetAdpcmComp, InitAdpcmComp
;*
;* Local constants
AdpcmPage .set      4          ; Adpcm variables are on page 4
;*
;*****
; .text
;*****
AdpcmComp:
;* This routine converts the input 14-bit linear PCM value into a 4-bit
;* Adpcm value.
;*
;* Assumes:
;*   DP = 0      Data page containing the interface variables
;*   SSXM        Sign extension mode must be on
;*
;* Passed:
;*   LinVal      14-bit linear value to be compressed (right-just)
;*
;* Returned:
;*   AdpcmVal    4-bit Adpcm value (right-just)
;*
;* Modified:
;*   ARP        Used to point to the AR0 temporary register
;*****
;* First, load the linear value to the local data space
;*
;   LAC    LinVal      ; Load the linear value
;   LDPK   AdpcmPage    ; Change to Adpcm variables
;   SACL   Sample       ; Store the linear sample in a local variable
;
;*****

```

SigDif:

```

;* Compute the partial signal estimate (Sez), the full signal estimate (Se),
;* and the quantizer scale factor (Y) from the previous sample's data. Then
;* compute the difference signal (D).
;*****
;*
;* Compute Sez, the partial signal estimate as:
;*    $Sez(k) = B1(k-1)*Dq(k-1) + \dots + B6(k-1)*Dq(k-6)$ 
;* Note: B1 (16b TC Q14), Dqi (16b TC Q0), Sez (16b TC Q0)
;*
;   ZAC                                ; Zero accumulator to start calculation
;   LT      Dq5                        ; Step through parameters, multiplying and
;   MPY     B6                         ; accumulating in reverse order. Shift
;   LTD     Dq4                        ; Dqi's up 1 location after multiplication
;   MPY     B5                         ; in preparation for calculation on next
;   LTD     Dq3                        ; sample.
;   MPY     B4
;   LTD     Dq2
;   MPY     B3
;   LTD     Dq1
;   MPY     B2
;   LTD     Dq
;   MPY     B1
;
;   APAC                                ; Get Q14 result in 32 bit accumulator
;   SACH    Sez,2                      ; Save as Q0
;
;* Now compute Se, the full signal estimate as:
;*    $Se(k) = A1(k-1)*Sr(k-1) + A2(k-1)*Sr(k-2) + Sez(k)$ 
;* Note: A1 (16b TC Q14), Sri (16b TC Q0), Se (16b TC Q0)
;*
;   LAC     Sez,14                    ; Load Sez as Q14 in accumulator
;   LT      Srl                      ; Add components from A1 and A2, and shift
;   MPY     A2                        ; Sr into Srl for calculation on next
;   LTD     Sr                        ; sample.
;   MPY     A1
;   APAC                                ; Get Q14 result in accumulator
;   SACH    Se,2                      ; Save as Q0
;
;* Calculate the limited speed control parameter (Al) from the previous
;* unlimited speed control parameter (Ap).
;*    $Al = 1 \text{ (64)} \quad \text{if } Ap > 1 \text{ (256)}$ 
;*    $Al = Ap \quad \text{if } Ap \leq 1$ 
;* Note: Al (7b POS Q6), Ap (10b POS Q8)
;*
;   LAC     One,6
;   SACL    Al                        ; Store 1 (Q6) in Al
;   LAC     Ap
;   SUB     One,8                    ; Check if Ap > 1 (Q8)
;   BGEZ    GoodAl                  ; Ap > 1, so keep 1 in Al
;   LAC     Ap,14                    ; Ap <= 1, so load in Q6 form in AcCh
;   SACH    Al                      ; Save modified Al
;
;GoodAl:
;*
;* Now compute Y, the adaptive quantizer scale factor as:
;*    $Y(k) = [1 - A1(k)]*Y1(k-1) + A1(k)*Yu(k-1)$ 
;*    $= A1(k)*[Yu(k-1) - Y1(k-1)] + Y1(k-1)$ 
;* Note: Al (7b POS Q6), Y1 (19b POS Q15), Yu (13b POS Q9), Y (13b POS Q9)
;*   Y1 has both a low (Y1l) and high (Y1h) word.
;*
;   LAC     Y1l,10                  ; First convert Y1 to (13b POS Q9) by

```

```

SACH Temp3      ; dropping least significant 6 bits
LAC Temp3      ; and right justifying result.
AND M1023      ; Mask off 10 bits of Y11 (for sign extension)
ADD Y1h,10
SACL Temp3      ; Store (13b POS Q9) value of Y1 in Temp3
LAC Yu
SUB Temp3      ; Get Yu - Y1 in accumulator
SACL Temp3      ; Store (13b POS Q9) value of Yu - Y1
ZALH Y1h      ; Load (19b POS Q15) value of Y1 to accum
ADDS Y11
LT A1          ; Multiply A1*(Yu - Y1), result is Q15
MPY Temp3
APAC          ; Add to Q15 Y1 in accum (accum = Y in Q15)
SACL Temp3      ; Now, convert Y to (13b POS Q9)
SACH Temp2
LAC Temp3,10   ; Drop 6 bits off of low word of Y
SACH Temp3
LAC Temp3      ; Right justify in accumulator
AND M1023      ; Mask 10 desired bits (remove sign extension)
ADD Temp2,10   ; Add top portion of Y
AND M8191      ; Mask 13 desired bits of result
SACL Y         ; Save (13b POS Q9) result in Y
LAC Y,14
SACH Temp3      ; Save (11b POS Q7) version of Y in Temp3
;
; * Now compute D, the difference signal as:
; *  $D(k) = S1(k) - Se(k)$ , where  $S1(k)$  is linear version of input sample
; * Note: All D, S1, and Se are (16b TC Q0)
;
LAC Sample     ; Load the current linear sample
SUB Se         ; Subtract signal estimate
SACL D         ; Store difference signal
;
; *****
; Aquan:
; * Adaptive quantization section. From the previously computed difference
; * signal (D), and adaptive quantizer scale factor (Y), compute the 4-bit
; * quantized difference signal (I).
; * *****
; * First get the log base 2 of the difference signal. This can be simplified
; * by using the approximation that  $\log_2(1 + x) = x$  for small x. Therefore,
; * determine the number of shifts required until  $D = 1.x$  (the exponent), and
; * then store x in a Q7 format, with the 4 bits of exponent above the
; * 'mantissa'.
;
ABS            ; Convert to positive value for log
SACL Temp1     ; Store difference in Temp1
;
; * Now, do binary search for left-most 1 to get exponent. Store exponent in
; * AR0, and load accumulator with D in Q7 format.
;
GetExp: SUB One,8      ; D - 256, exp = 0-14
      BGEZ C8To14
C0To7: ADD M15,4       ; D - 16, exp = 0-7
      BGEZ C4To7
C0To3: ADD Three,2     ; D - 4, exp = 0-3
      BGEZ C2To3
C0To1: ADD One,1       ; D - 2, exp = 0-1
      BGEZ Exp1

```

```

Exp0: LARK 0,0      ; exp = 0
      LAC Temp1,7
      B GetMan
Exp1: LARK 0,1      ; exp = 1
      LAC Temp1,6
      B GetMan
C2To3: SUB One,2    ; D - 8, exp = 2-3
      BGEZ Exp3
Exp2: LARK 0,2      ; exp = 2
      LAC Temp1,5
      B GetMan
Exp3: LARK 0,3      ; exp = 3
      LAC Temp1,4
      B GetMan
C4To7: SUB Three,4  ; D - 64, exp = 4-7
      BGEZ C6To7
C4To5: ADD One,5    ; D - 32, exp = 4-5
      BGEZ Exp5
Exp4: LARK 0,4      ; exp = 4
      LAC Temp1,3
      B GetMan
Exp5: LARK 0,5      ; exp = 5
      LAC Temp1,2
      B GetMan
C6To7: SUB One,6    ; D - 128, exp = 6-7
      BGEZ Exp7
Exp6: LARK 0,6      ; exp = 6
      LAC Temp1,1
      B GetMan
Exp7: LARK 0,7      ; exp = 7
      LAC Temp1
      B GetMan
C8To14: SUB M15,8   ; D - 4096, exp = 8-14
      BGEZ C12To14
C8To11: ADD Three,10 ; D - 1024, exp = 8-11
      BGEZ C10To11
C8To9: ADD One,9    ; D - 512, exp = 8-9
      BGEZ Exp9
Exp8: LARK 0,8      ; exp = 8
      LAC Temp1,15
      SACH Temp1
      LAC Temp1
      B GetMan
Exp9: LARK 0,9      ; exp = 9
      LAC Temp1,14
      SACH Temp1
      LAC Temp1
      B GetMan
C10To11: SUB One,10 ; D - 2048, exp = 10-11
      BGEZ Exp11
Exp10: LARK 0,10    ; exp = 10
      LAC Temp1,13
      SACH Temp1
      LAC Temp1
      B GetMan
Exp11: LARK 0,11    ; exp = 11
      LAC Temp1,12
      SACH Temp1
      LAC Temp1
      B GetMan

```

```

C12To14: SUB    Three,12    ; D - 16384, exp = 12-14
          BGEZ    Exp14
C12To13: ADD    One,13     ; D - 8192, exp = 13-14
          BGEZ    Exp13
Exp12:   LARK    0,12      ; exp = 12
          LAC     Temp1,11
          SACH    Temp1
          LAC     Temp1
          B       GetMan
Exp13:   LARK    0,13      ; exp = 13
          LAC     Temp1,10
          SACH    Temp1
          LAC     Temp1
          B       GetMan
Exp14:   LARK    0,14      ; exp = 14
          LAC     Temp1,9
          SACH    Temp1
          LAC     Temp1

;*
;* Now, AR0 contains the exponent, and accumulator has the Q7 mantissa
;* Store Log2(D) as 0000 0EEE EMMM MMMM (Q7).
;*
GetMan:
  AND     M127          ; Mask off 7 bits (Q7) (remove leading 1)
  SAR     0,Temp1       ; Store the exponent ...
  ADD     Temp1,7        ; and add as integer part above Q7 decimal
;*
;* Now, subtract the scale factor from log2(D). An offset is added to avoid
;* negative numbers.
;*
  ADD     One,11         ; Add offset, 2048 is now 0
  SUB     Temp3
  AND     M4095          ; Temp3 has Y in (11b POS Q7) format
                      ; Mask off low 12 bits
;*
;* Now, quantize the scaled difference signal with a 4 bit quantizer.
;* The table values are as follows (12b Q7): (note that 2048 = 0)
;*
;* I = 0      acc < 2041      acc < -0.055
;* I = 1      2041 <= acc < 2171    -0.055 <= acc < 0.961
;* I = 2      2171 <= acc < 2250    0.961 <= acc < 1.578
;* I = 3      2250 <= acc < 2309    1.578 <= acc < 2.039
;* I = 4      2309 <= acc < 2358    2.039 <= acc < 2.422
;* I = 5      2358 <= acc < 2404    2.422 <= acc < 2.781
;* I = 6      2404 <= acc < 2453    2.781 <= acc < 3.164
;* I = 7      2453 <= acc
;*
;* Do binary search for table entry
;*
  SUB     K2309          ; acc - 2309, I = 0-7
  BGEZ    CI4To7
CI10To3: ADD    K138      ; acc - 2171, I = 0-3
  BGEZ    CI2To3
CI10To1: ADD    K130      ; acc - 2041, I = 0-1
  BGEZ    Ieq1
Ieq0:   LACK    0         ; I = 0
          B       HaveIMag
Ieq1:   LACK    1         ; I = 1
          B       HaveIMag
CI2To3: SUB     K79        ; acc - 2250, I = 2-3
  BGEZ    Ieq3
Ieq2:   LACK    2         ; I = 2

```

```

          B       HaveIMag
Ieq3:   LACK    3         ; I = 3
          B       HaveIMag
CI4To7: SUB     K95        ; acc - 2404, I = 4-7
  BGEZ    CI6To7
CI5To6: ADD     K46        ; acc - 2358, I = 5-6
  BGEZ    Ieq5
Ieq4:   LACK    4         ; I = 4
          B       HaveIMag
Ieq5:   LACK    5         ; I = 5
          B       HaveIMag
CI6To7: SUB     K49        ; acc - 2453, I = 6-7
  BGEZ    Ieq7
Ieq6:   LACK    6         ; I = 6
          B       HaveIMag
Ieq7:   LACK    7         ; I = 7
          B       HaveIMag
HaveIMag:
;*
;* Accumulator now holds the magnitude of the quantized difference I
;*
  SACL    IM             ; Save the quantized difference magnitude
  SACL    I              ; Save as quantized difference
  LAC     D              ; Load the linear difference
  BGEZ    HaveADPCM      ; I is ok if D >= 0
  LAC     IM             ; Need to invert bits of IM if D < 0
  XOR     M15            ; Invert only lower 4 bits (the rest are 0)
  SACL    I              ; Save the inverted quantized difference
HaveADPCM:
  LAC     I              ; Load the quantized Adpcm value
  LDPK    0              ; Access page 0 values
  SACL    AdpcmVal       ; Store the returned Adpcm value
  LDPK    AdpcmPage      ; Switch back to Adpcm variables
;*
*****
IAquan:
;* Inverse adaptive quantizer section. From the quantized difference signal
;* (I) and the scale factor (Y), compute the reconstructed difference
;* signal (Dq).
*****
;* First, convert quantized difference magnitude to Log domain
;*
  LAC     IM             ; Load the magnitude
  ADD     InvQTableOffset ; Add offset of inverse quantization table
  TBLR    Temp1          ; Read table value into Temp1 (Q7)
;*
;* Add back the scale factor
;*
  LAC     Temp1          ; Move [log2(Dq) - Y] value to accumulator
  ADD     Temp3          ; Add (11b POS Q7) version of Y
  AND     M2047          ; Mask lower 11 bits of result
  SACL    Temp2          ; Save log2(Dq) to Temp2
;*
;* Now compute Dq from log2(Dq). log2(Dq) is 0000 0EEE EMMM MMMM (Q7)
;* Dq is (16b TC Q0)
;* Required shifts are:
;* Exp  00 01 02 03 04 05 06 07 08 09 10 11 12 13 14
;* Shift 09 10 11 12 13 14 15 00 01 02 03 04 05 06 07
;* Where the result is in AccH for Exp = 0-6, and in AccL for Exp = 7-14
;* (Right shift for Exp = 0-6, Left shift for Exp = 7-14)

```

```

;*
LAC    Temp2,9      ; Load exponent into high accumulator ...
SACH   Temp1        ; and store in Temp1
LACK   127          ; Load mask for lower 7 bits ...
AND    Temp2        ; and mask lower 7 bits (mantissa)
ADD    One,7         ; Add 1 (Q7) to get 1+x
SACL   Temp2        ; Store mantissa
LT     Temp2        ; Prepare to shift
LAC    Temp1        ; Read shift multiplier corresponding to ...
ADD    ShiftTableOffset ; the exponent from a table
TBLR   Temp3        ;
MPY    Temp3        ; Multiply by shift constant
PAC    ;
BLZ    LeftShift    ; Branch if left shift was required
SACH   Dq,1         ; Right shift 1 more, store in Dq (mag)
B      AddSign      ; Branch to add sign
LeftShift:
ABS    ;
SACL   Dq           ; Save result in Dq (mag)
AddSign:
LAC    One,11       ; Save +1 (Q11) as Sgn(Dq)
SACL   SDq          ;
LAC    I            ; Check sign of I (4th bit)
SUB    One,3        ;
BLZ    HaveDq       ; Branch if I +ve, SDq and Dq are correct
ZAC    ;
SUB    Dq           ; Negate Dq, I is -ve
SACL   Dq           ; Store modified Dq
LAC    Minus,11     ;
SACL   SDq          ; Save -1 (Q11) as Sgn(Dq)
HaveDq:
;*
*****
QSPA:
;* Quantizer scale factor adaptation section. Using the previously obtained
;* quantized difference signal (I), compute the new scale factors (Yu, Y1).
*****
;*
;* First compute W[I] from the table
LAC    WTableOffset ; Load table offset
ADD    IM           ; Add magnitude of quantized difference
TBLR   Temp1        ; Lookup WI (Q4), store in Temp1
;*
;* Update the fast scale factor as:
;*  $Y_u(k) = (1 - 2^{*-5}) * Y(k) + (2^{*-5}) * W_I(k)$ 
;* Note: Yu and Y are (13b POS Q9), WI is (12b TC Q4 sign extended to 16b)
;*
LAC    Y,12         ; Load Y in Q21 form
SUB    Y,7          ; Subtract Y/32 (Q21)
ADD    Temp1,12      ; Add WI/32 (Q21)
SACH   Yu,4          ; Store Yu in (13b POS Q9) format
LAC    Yu           ; Reload Yu ...
AND    M8191        ; and mask lower 13 bits
SACL   Yu           ; Save final Yu
;*
;* Limit the fast scale factor  $1.06 \leq Y_u \leq 10.0$ 
;* Q9 form (544 <= Yu <= 5120)
;*
SUB    K544         ; Check low threshold
BGEZ   CheckHi      ; Branch if above lower threshold

```

```

LAC    K544         ; Load lower threshold, as it was below
B      StoreYu      ; Branch to store lower limit
CheckHi:
SUB    K4576        ; acc - 5120, check upper threshold
BLEZ   GoodYu       ; Branch if within limits
LAC    K5120        ; Load upper threshold, as it was above
StoreYu:
SACL   Yu           ; Store the modified Yu
GoodYu:
;*
;* Update the slow scale factor as:
;*  $Y_1(k) = (1 - 2^{*-6}) * Y_1(k-1) + (2^{*-6}) * Y_u(k)$ 
;* Note: Yu is (13b POS Q9), Y1 is (19b POS Q15)
;*
ZALH   Y1h          ;
ADDS   Y1l          ; Load Y1 to accumulator
RPTK   5            ;
SFL    Y1h          ; Shift Y1 left 6 bits (Q21)
SUBH   Y1h          ;
SUBS   Y1l          ; Subtract (2^{*-6}) * Y1 (Q21)
ADD    Yu,6         ; Add (2^{*-6}) * Yu (Q21)
RPTK   5            ;
SFR    Y1h          ; Shift Y1 right 6 bits (Q15)
SACL   Y1l          ;
SACH   Y1h          ; Save modified Y1
LACK   7            ; Mask 3 lower bits of Y1h
AND    Y1h          ;
SACL   Y1h          ;
;*
*****
SpeedControl:
;* Adaptation speed control section. Using the previously obtained quantized
;* signal magnitude (IM), compute the adaptation speed control (Ap).
*****
;*
;* First compute the F[I] function
;*
LAC    FITableOffset ; Look-up value of FI function
ADD    IM           ;
TBLR   Temp1        ; Load FI value into Temp1 (Q4)
;*
;* Now, update the short term average of FI as:
;*  $D_{ms}(k) = (1 - 2^{*-5}) * D_{ms}(k-1) + (2^{*-5}) * F_I(k)$ 
;* Note: FI is (7b POS Q4), Dms is (12b POS Q9)
;*
LAC    Temp1,15     ; Load (2^{*-5}) * FI (Q24)
ADD    Dms,15       ; Add Dms(k-1) (Q24)
SUB    Dms,10       ; Subtract (2^{*-5}) * Dms (Q24)
SACH   Dms,1        ; Save the new Dms (Q9)
;*
;* Update long term average of FI as:
;*  $D_{ml}(k) = (1 - 2^{*-7}) * D_{ml}(k-1) + (2^{*-7}) * F_I(k)$ 
;* Note: FI is (7b POS Q4), Dml is (14b POS Q11)
;*
LAC    Temp1,15     ; Load (2^{*-7}) * FI (Q26)
ADD    Dml,15       ; Add Dml(k-1) (Q26)
SUB    Dml,8        ; Subtract (2^{*-7}) * Dml (Q26)
SACH   Dml,1        ; Save the new Dml (Q11)
;*
;* Now compute the speed control parameter as:

```

```

;*      Ap(k) = (1 - 2**(-4))*Ap(k-1) + (2**(-3))
;*              if (Y < 3) or (|Dms - Dml| > (2**(-3))*Dml)
;*              - (1 - 2**(-4))*Ap(k-1)
;*              otherwise
;* Note: Dms is (12b POS Q9), Dml is (14b POS Q11), Ap is (10b POS Q8)
;*
ZALH  Ap          ; Load Ap (Q24)
SUB   Ap,12       ; Subtract (2**(-4))*Ap (Q24)
SACH  Ap          ; Save (1 - 2**(-4))*Ap (Q8)
LAC   Y
SUB   Three,9     ; Check if Y < 3 (Q9)
BLZ   AddEighth   ; Add 1/8 if so.
ZALH  Dms         ; Load Dms (Q25)
SUB   Dml,14      ; Subtract Dml (Q25)
ABS   Dml,11      ; Get magnitude of result
SUB   Dml,11      ; Check if greater than Dml/8 (Q25)
BLZ   GoodAp      ; Branch if not
AddEighth:
LAC   Ap          ; Reload Ap
ADD   One,5       ; Add 1/8 (Q8)
SACL  Ap          ; Save modified Ap
GoodAp:
;*
*****
APred:
;* Adaptive predictor section. Using the previously obtained difference
;* signal (Dq), and the previous signal estimate (Se), compute the new
;* adaptive filter coefficients.
*****
;* First, compute the reconstructed signal (Sr) as:
;*      Sr(k) = Dq(k) + Se(k)
;* Note: Dq, Se, and Sr are (16b TC Q0)
;*
LAC   Dq
ADD   Se
SACL  Sr          ; Save Sr = Dq + Se (Q0)
;*
;* Compute coefficients of 6th order FIR predictor filter
;*      Bi(k) = (1 - 2**(-8))*Bi(k-1) + (2**(-7))*sgn(Dq(k))*sgn(Dq(k-1))
;*      for i = 1 .. 6 (Bi implicitly limited to +/- 2)
;* Note: Bi are (16b TC Q14), SDqi are (16b TC Q11; +1 = +2048, -1 = -2048)
;*
GetB6: LT   SDq6          ; Load old Bi (Q29)
LAC   B6,15          ; Load old Bi (Q29)
SUB   B6,7           ; Subtract (2**(-8))*Bi (Q29)
MPY   SDq            ; SDq * SDq6 * (2**(-7)) (Q29)
LTD   SDq5           ; Create new Bi, shift data for next sample
SACH  B6,1          ; Save new Bi in (Q14) format
GetB5: LAC   B5,15
SUB   B5,7
MPY   SDq
LTD   SDq4
SACH  B5,1
GetB4: LAC   B4,15
SUB   B4,7
MPY   SDq
LTD   SDq3
SACH  B4,1
GetB3: LAC   B3,15

```

```

SUB   B3,7
MPY   SDq
LTD   SDq2
SACH  B3,1
GetB2: LAC   B2,15
SUB   B2,7
MPY   SDq
LTD   SDq1
SACH  B2,1
GetB1: LAC   B1,15
SUB   B1,7
MPY   SDq
LTD   SDq
SACH  B1,1
;*
;* Update coefficients of 2nd order IIR prediction filter
;*
;* First get sign of the sum of difference signal (Dq), and the partial signal
;* estimate (Sez).
;* Note: Dq and Sez are (16b TC Q0), Pk1 is (10b TC Q9 sign extended to 16b)
;*
DMOV  Pk1          ; Shift Pk1-->Pk2
DMOV  Pk0          ; Shift Pk0-->Pk1
LAC   One,9
SACL  Pk0          ; Store +1 (Q9) as initial sign
LAC   Sez
ADD   Dq           ; Calculate (Sez + Dq)
BGEZ  GoodPk       ; Branch if result is +ve, already saved +1
LAC   Minus,9
SACL  Pk0          ; Result is -ve, save -1 (Q9)
GoodPk:
LT    Pk0          ; Load Pk0 to T-reg in prep. for multiply
;*
;* Now calculate f[A1(k-1)] as:
;*      f[A1(k-1)] = 4*A1, if |A1| <= 1/2
;*                  = 2*sgn(A1), if |A1| > 1/2
;* Note: A1 is (16b TC Q14), F is (16b TC Q13)
;*
LAC   A1,1         ; Load 2*A1 (Q14) to accum
SACL  Temp3        ; Save as initial F/2
BLZ   NegA1        ; Branch if A1 is negative
SUB   One,14       ; Calculate 2*(A1 - 1/2) (Q14)
BLZ   HaveF        ; F/2 is ok if |A1| < 1/2
LAC   One,14       ; |A1| > 1/2, so load limit 1 (Q14)
B     SaveF
NegA1:
ABS   One,14       ; Make A1 positive
SUB   One,14       ; Check if |A1| < 1/2
BLZ   HaveF        ; F/2 is ok if so
LAC   Minus,14     ; |A1| > 1/2, so load limit -1 (Q14)
SaveF:
SACL  Temp3        ; Save modified F/2
HaveF:
;*
;* Now calculate A1(k) as:
;*      A1(k) = (1 - 2**(-8))*A1(k-1) + 3*(2**(-8))*sgn[P(k)]*sgn[P(k-1)]
;* Note: A1 is (16b TC Q14), sgn(p) is (10b TC Q9 sign extended to 16b)
;*
LAC   A1,12        ; Load A1 (Q26)
SUB   A1,4          ; Subtract (2**(-8))*A1 (Q26)

```



```

MPY    Pk1          ; Calculate sgn(Pk1)*sgn(Pk), sgn(Pk) is ...
APAC                   ; in T register already. Result is (Q18)
APAC
APAC
SACH    A1,4         ; Add 3*(2**-8)*sgn(Pk1)*sgn(Pk) (Q26)
PAC          ; Save new A1 in (Q14)
          ; Acc has sgn[p(k-1)]*sgn[p(k)]
;*
;* Now calculate A2(k) as:
;*   A2(k) = (1 - 2**-7)*A2(k-1) +
;*   + (2**-7)*{sgn[P(k)]*sgn[P(k-2)]-f[A1(k-1)]*sgn[P(k)]*sgn[P(k-1)]}
;* Note: A2 is (16b TC Q14), sgn(p) is (10b TC Q9), F is (16b TC Q13)
;*
BGEZ    SubtractF    ; If sgn(Pk1)*sgn(Pk) +ve then subtract F
ZAC          ; Otherwise, negate F then subtract
SUB      Temp3        ; Temp3 has F/2
SACL     Temp3        ; Save negated F/2
SubtractF:
LAC      A2,12        ; Load old A2 (Q26)
SUB      A2,5         ; Subtract (2**-7)*A2 (Q26)
MPY      Pk2          ; Calculate sgn(Pk)*sgn(Pk2) (Q18)
APAC
APAC          ; Add (2**-7)*sgn(Pk)*sgn(Pk2) (Q26)
SUB      Temp3,6      ; Subtract (2**-7)*F (Q26)
SACH     A2,4         ; Save new A2 (Q14)
;*
;* Now limit A2 to +/- 0.75, and prevent overflow.
;* Note: A2 is (16b TC Q14)
;*
LAC      A2
ABS          ; Take absolute value of A2
SUB      Three,12     ; Check that |A2| < 0.75 (Q14)
BLEZ     GoodA2       ; Branch if A2 is within limits
LAC      A2
BGEZ     A2PosLimit   ; Save +ve limit if A2 is +ve
A2NegLimit:
LAC      Minus3,12    ; Save -ve limit of -0.75 (Q14)
B        SaveA2
A2PosLimit:
LAC      Three,12     ; Save +ve limit of +0.75 (Q14)
SaveA2:
SACL     A2           ; Save limited A2
GoodA2:
;*
;* Now, limit A1 to +/- (1 - 2**-4 - A2)
;* Note: A1 and A2 are (16b TC Q14)
;*
LAC      M15,10       ; Load (1 - 2**-4) (Q14)
SUB      A2           ; Subtract A2 (Q14)
SACL     Temp1        ; Save limit of (1 - 2**-4 - A2) in Temp1
LAC      A1
ABS          ; Load |A1|
SUB      Temp1        ; Compare with limit
BLEZ     GoodA1       ; Branch if |A1| <= limit
LAC      A1
BGEZ     LimitA1      ; Save positive limit if A1 > 0
ZAC
SUB      Temp1        ; Calculate negative limit
SACL     Temp1        ; Save negative limit
LimitA1:
LAC      Temp1        ; Load limit value to accumulator

```

```

SACL     A1           ; Store limited A1 value
GoodA1:
;*
;*****
;* This pass of the encoding routine is done. Return to calling routine.
;*
LDPK     0           ; Reset to page 0 variables
RET          ; Return to calling routine
;*
;*****
InitAdpcmComp:
;* Routine to initialize the constants used by the ADPCM algorithm, and
;* reset the ADPCM algorithm variables. This routine must be called before
;* the first Adpcm conversion.
;*
;* Assumes:
;* DP = 0           DP is set to page 0 upon exit
;* ?SXM             Sign extension can be either on or off
;*****
LDPK     AdpcmPage    ; Switch to Adpcm variables
;*
;* Define basic constants
;*
LACK     1
SACL     One
SUB      One,1
SACL     Minus
LACK     3
SACL     Three
SUB      Three,1
SACL     Minus3
;*
;* Define bit masks
;*
LACK     15
SACL     M15
LACK     127
SACL     M127
LAC      One,10
SUB      One
SACL     M1023
ADD      One,10
SACL     M2047
ADD      One,11
SACL     M4095
ADD      One,12
SACL     M8191
;*
;* Define ADPCM specific constants
LALK     544
SACL     K544
LALK     5120
SACL     K5120
LALK     4576
SACL     K4576
LALK     2309
SACL     K2309
LACK     138

```

```

SACL K138
LACK 130
SACL K130
LACK 79
SACL K79
LACK 95
SACL K95
LACK 46
SACL K46
LACK 49
SACL K49
;*
;* Initialize data table pointers
;*
LT One
MPYK InvQTable
PAC
SACL InvQTableOffset
MPYK ShiftTable
PAC
SACL ShiftTableOffset
MPYK WITable
PAC
SACL WITableOffset
MPYK FITable
PAC
SACL FITableOffset
;*
;* Fall through to ResetAdpcmComp to reset the ADPCM variables
;*
*****
ResetAdpcmComp:
;* Routine to reset the ADPCM compression algorithm variables.
;*
;* Assumes:
;* DP = 0 DP is set to page 0 upon exit
;* ?SXM Sign extension can be either on or off
*****
LDPK AdpcmPage ; Switch to Adpcm variables
;*
;* Zero the predictor coefficients, and delayed values
;*
ZAC
SACL Dq
SACL Dq1
SACL Dq2
SACL Dq3
SACL Dq4
SACL Dq5
SACL B1
SACL B2
SACL B3
SACL B4
SACL B5
SACL B6
SACL Ap
SACL Sr
SACL Srl
SACL A1

SACL A2
SACL Dms
SACL Dml
SACL Ylh
;*
;* Set default values for the scale factors
;*
LAC K544
SACL Yu
LAC One,15
ADD One,11
SACL Yll
;*
;* Initialize the signs of delayed values
;*
LAC One,11
SACL SDq1
SACL SDq2
SACL SDq3
SACL SDq4
SACL SDq5
SACL SDq6
;*
LAC One,9
SACL Pk0
SACL Pk1
;*
LDPK 0 ; Switch back to page 0 variables
RET
*****
.data
*****
;*
;* Initialized data tables
;*
;* Inverse quantization table
InvQTable: ; (16b TC Q7)
.word 65401 ; |I| = 0 (-1.05)
.word 68 ; |I| = 1 (0.53)
.word 165 ; |I| = 2 (1.29)
.word 232 ; |I| = 3 (1.81)
.word 285 ; |I| = 4 (2.23)
.word 332 ; |I| = 5 (2.59)
.word 377 ; |I| = 6 (2.95)
.word 428 ; |I| = 7 (3.34)
;*
;* Log to linear shift factor table
ShiftTable: ; (16b TC Q0)
.word 256 ; Exp = 0 (Shift 9) - 1
.word 512 ; Exp = 1 (Shift 10) - 1
.word 1024 ; Exp = 2 (Shift 11) - 1
.word 2048 ; Exp = 3 (Shift 12) - 1
.word 4096 ; Exp = 4 (Shift 13) - 1
.word 8192 ; Exp = 5 (Shift 14) - 1
.word 16384 ; Exp = 6 (Shift 15) - 1
.word -1 ; Exp = 7 (Shift 0)
.word -2 ; Exp = 8 (Shift 1)
.word -4 ; Exp = 9 (Shift 2)

```

```

.word -8      ; Exp = 10 (Shift 3)
.word -16     ; Exp = 11 (Shift 4)
.word -32     ; Exp = 12 (Shift 5)
.word -64     ; Exp = 13 (Shift 6)
.word -128    ; Exp = 14 (Shift 7)

```

```

;* Scale factor multiplier table

```

```

WITable:      ; (16b TC Q4)
.word 65524   ; |I| = 0 (-0.75)
.word 4       ; |I| = 1 (0.25)
.word 27      ; |I| = 2 (1.69)
.word 50      ; |I| = 3 (3.12)
.word 98      ; |I| = 4 (6.12)
.word 184     ; |I| = 5 (11.50)
.word 340     ; |I| = 6 (21.25)
.word 1108    ; |I| = 7 (69.25)

```

```

;* Rate of change weighting function table

```

```

FITable:      ; (16b POS Q4)
.word 0       ; |I| = 0 (0)
.word 0       ; |I| = 1 (0)
.word 0       ; |I| = 2 (0)
.word 16      ; |I| = 3 (1)
.word 16      ; |I| = 4 (1)
.word 16      ; |I| = 5 (1)
.word 48      ; |I| = 6 (3)
.word 112     ; |I| = 7 (7)

```

```

*****

```

```

;* Variable declarations

```

```

.bss A1,1      ; Coefficients for 2nd order IIR predictor
.bss A2,1      ; ...
.bss B1,1      ; Coefficients for 6th order FIR predictor
.bss B2,1      ; ...
.bss B3,1      ; ...
.bss B4,1      ; ...
.bss B5,1      ; ...
.bss B6,1      ; ...
.bss Dq,1      ; Inverse quantized difference signal
.bss Dq1,1     ; Delayed versions of Dq
.bss Dq2,1     ; ...
.bss Dq3,1     ; ...
.bss Dq4,1     ; ...
.bss Dq5,1     ; ...
.bss SDq,1     ; Sign of inverse quantized difference signal
.bss SDq1,1    ; Delayed versions of SDq
.bss SDq2,1    ; ...
.bss SDq3,1    ; ...
.bss SDq4,1    ; ...
.bss SDq5,1    ; ...
.bss SDq6,1    ; ...
.bss Sr,1      ; Reconstructed signal estimate
.bss Sr1,1     ; Delayed version of Sr
.bss Pk0,1     ; Sign of partial reconstructed signal
.bss Pk1,1     ; Delayed versions of Pk0
.bss Pk2,1     ; ...
.bss Se,1      ; Signal estimate
.bss Sez,1     ; Partial signal estimate

```

```

.bss Y11,1     ; Low word of unlocked scale factor
.bss Y1h,1     ; High word of unlocked scale factor
.bss Yu,1      ; Locked scale factor
.bss Y,1       ; Adaptive quantizer scale factor
.bss Dms,1     ; Short term average of ADPCM signal
.bss Dml,1     ; Long term average of ADPCM signal
.bss Ap,1      ; Unlimited speed control factor
.bss Al,1      ; Limited speed control factor
.bss I,1       ; Quantized difference signal
.bss IM,1      ; Quantized difference signal magnitude
.bss D,1       ; Difference signal
.bss Temp1,1   ; Temporary scratch variables
.bss Temp2,1   ; ...
.bss Temp3,1   ; ...
.bss Sample,1  ; Input linear speech sample

```

```

;*
;* Reserve space for constants

```

```

.bss InvQTableOffset,1 ; Offset to Inverse Quantization table
.bss ShiftTableOffset,1 ; Offset to Log-to-Linear shift table
.bss WITableOffset,1   ; Offset to Scale Factor Multiplier table
.bss FITableOffset,1   ; Offset to Rate-of-Change Weighting table
.bss One,1             ; 1
.bss Three,1           ; 3
.bss Minus,1           ; -1
.bss Minus3,1          ; -3
.bss M15,1             ; 15 (000F) 4 bit mask
.bss M127,1            ; 127 (007F) 7 bit mask
.bss M1023,1           ; 1023 (03FF) 10 bit mask
.bss M2047,1           ; 2047 (07FF) 11 bit mask
.bss M4095,1           ; 4095 (0FFF) 12 bit mask
.bss M8191,1           ; 8191 (3FFF) 13 bit mask
.bss K2309,1           ; 2309
.bss K138,1            ; 138
.bss K130,1            ; 130
.bss K79,1             ; 79
.bss K95,1             ; 95
.bss K46,1             ; 46
.bss K49,1             ; 49
.bss K544,1            ; 544
.bss K4576,1           ; 4576
.bss K5120,1           ; 5120

```

```

;*
*****
.end
*****

```

AdpcmExp.asm

```

;*
;* AdpcmExp.asm
;* May 11, 1990
;* Chris Rose
;*
;* Routine to perform 32Kbps ADPCM decoding. The algorithm used is modelled
;* after the CCITT standard, but is not identically bit compatible. The
;* algorithm is taken from "32-kbit/s ADPCM with the TMS32010", an application
;* note from Texas Instruments. The original source code was modified slightly

```

```

; * to be incorporated into the silence compression program.
; *
; * Number formats:
; *   nb      - number is stored in n bits
; *   Qn      - there are n decimal bits
; *   TC      - two's complement notation
; *   POS     - unsigned notation
; * *****
; * External references
; *   .def  AdpcmExp, InitAdpcmExp, ResetAdpcmExp
; *   .ref  LinVal, MuLawVal, AdpcmVal
; *
; * Local constants
AdpcmPage .set 4 ; Page containing Adpcm variables
; *
; * *****
; * .text
; * *****
AdpcmExp:
; * This routine converts the input 4-bit Adpcm value into a 14-bit linear
; * PCM value.
; *
; * Assumes:
; *   DP = 0      Data page containing interface variables
; *   SSXM       Sign extension mode must be on to handle 16-bit -ve numbers
; *
; * Passed:
; *   AdpcmVal    4-bit Adpcm value to be expanded by this routine (right-just)
; *
; * Returned:
; *   LinVal      Linear, reconstructed PCM signal (right-just)
; *
; * Modified:
; *   ARP         Used to point to the AR0 temporary register
; * *****
; *   LAC  AdpcmVal      ; Load the Adpcm value
; *   LDPK AdpcmPage     ; Switch to Adpcm variables
; *   SACL I             ; Save the Adpcm value as a local variable
; * *****
; * SigDif:
; * Compute the partial signal estimate (Sez), the full signal estimate (Se),
; * and the quantizer scale factor (Y) from the previous sample's data.
; * *****
; * Compute Sez, the partial signal estimate as:
; *    $Sez(k) = B1(k-1)*Dq(k-1) + \dots + B6(k-1)*Dq(k-6)$ 
; * Note: B1 (16b TC Q14), Dq1 (16b TC Q0), Sez (16b TC Q0)
; *
; *   ZAC          ; Zero accumulator to start calculation
; *   LT  Dq5      ; Step through parameters, multiplying and
; *   MPY B6       ; accumulating in reverse order. Shift
; *   LTD Dq4      ; Dq1's up 1 location after multiplication
; *   MPY B5       ; in preparation for calculation on next
; *   LTD Dq3      ; sample.
; *   MPY B4
; *   LTD Dq2
; *   MPY B3

```

```

LTD  Dq1
MPY  B2
LTD  Dq
MPY  B1
APAC                      ; Get Q14 result in 32 bit accumulator
SACH Sez,2                ; Save as Q0
; *
; * Now compute Se, the full signal estimate as:
; *    $Se(k) = A1(k-1)*Sr(k-1) + A2(k-1)*Sr(k-2) + Sez(k)$ 
; * Note: A1 (16b TC Q14), Sri (16b TC Q0), Se (16b TC Q0)
; *
; *   LAC  Sez,14         ; Load Sez as Q14 in accumulator
; *   LT   Sri            ; Add components from A1 and A2, and shift
; *   MPY  A2             ; Sr into Sri for calculation on next
; *   LTD  Sr            ; sample.
; *   MPY  A1
; *   APAC                      ; Get Q14 result in accumulator
; *   SACH Se,2          ; Save as Q0
; *
; * Calculate the limited speed control parameter (Al) from the previous
; * unlimited speed control parameter (Ap).
; *   Al = 1 (64) if Ap > 1 (256)
; *   Al = Ap    if Ap <= 1
; * Note: Al (7b POS Q6), Ap (10b POS Q8)
; *
; *   LAC  One,6          ; Store 1 (Q6) in Al
; *   SACL Al             ;
; *   LAC  Ap             ;
; *   SUB  One,8          ; Check if Ap > 1 (Q8)
; *   BGEZ GoodAl         ; Ap > 1, so keep 1 in Al
; *   LAC  Ap,14         ; Ap <= 1, so load in Q6 form in AcCh
; *   SACH Al            ; Save modified Al
; *
; * GoodAl:
; *
; * Now compute Y, the adaptive quantizer scale factor as:
; *    $Y(k) = [1 - Al(k)]*Yl(k-1) + Al(k)*Yu(k-1)$ 
; *    $= Al(k)*(Yu(k-1) - Yl(k-1)) + Yl(k-1)$ 
; * Note: Al (7b POS Q6), Yl (19b POS Q15), Yu (13b POS Q9), Y (13b POS Q9)
; *   Yl has both a low (Yll) and high (Ylh) word.
; *
; *   LAC  Yll,10         ; First convert Yl to (13b POS Q9) by
; *   SACH Temp3          ; dropping least significant 6 bits
; *   LAC  Temp3          ; and right justifying result.
; *   AND  M1023          ; Mask off 10 bits of Yll (for sign extension)
; *   ADD  Ylh,10         ;
; *   SACL Temp3          ; Store (13b POS Q9) value of Yl in Temp3
; *   LAC  Yu             ;
; *   SUB  Temp3          ; Get Yu - Yl in accumulator
; *   SACL Temp3          ; Store (13b POS Q9) value of Yu - Yl
; *   ZALH Ylh           ; Load (19b POS Q15) value of Yl to accum
; *   ADDS Yll           ;
; *   LT   Al            ; Multiply Al*(Yu - Yl), result is Q15
; *   MPY  Temp3          ;
; *   APAC                      ; Add to Q15 Yl in accum (accum = Y in Q15)
; *   SACL Temp3          ; Now, convert Y to (13b POS Q9)
; *   SACH Temp2          ;
; *   LAC  Temp3,10       ; Drop 6 bits off of low word of Y
; *   SACH Temp3          ;
; *   LAC  Temp3          ; Right justify in accumulator
; *   AND  M1023          ; Mask 10 desired bits (remove sign extension)

```

```

ADD    Temp2,10      ; Add top portion of Y
AND    M8191         ; Mask 13 desired bits of result
SACL   Y             ; Save (13b POS Q9) result in Y
LAC    Y,14
SACH   Temp3         ; Save (11b POS Q7) version of Y in Temp3
;*****
;Aquau:
;* Inverse adaptive quantizer section. From the quantized difference signal
;* (I) and the scale factor (Y), compute the reconstructed difference
;* signal (Dq).
;*****
;* First, convert quantized difference magnitude to Log domain
;*
LACK   7
AND    I             ; Mask off lower 3 magnitude bits of I
SACL   IM            ; Save the magnitude (assume positive)
LAC    I
SUB    One,3         ; Check sign bit of I
BLZ    GoodIM        ; Branch if positive, IM is good
LACK   7
XOR    IM            ; Invert magnitude bits, I was negative
SACL   IM            ; Save modified IM
GoodIM:
LAC    IM            ; Load the magnitude
ADD    InvQTableOffset ; Add offset of inverse quantization table
TBLR   Temp1         ; Read table value into Temp1 (Q7)
;*
;* Add back the scale factor
;*
LAC    Temp1         ; Move [log2(Dq) - Y] value to accumulator
ADD    Temp3         ; Add (11b POS Q7) version of Y
AND    M2047         ; Mask lower 11 bits of result
SACL   Temp2         ; Save log2(Dq) to Temp2
;*
;* Now compute Dq from log2(Dq). log2(Dq) is 0000 0EEE EMMM MMM (Q7)
;* Dq is (16b TC Q0)
;* Required shifts are:
;* Exp  00 01 02 03 04 05 06 07 08 09 10 11 12 13 14
;* Shift 09 10 11 12 13 14 15 00 01 02 03 04 05 06 07
;* Where the result is in AccH for Exp = 0-6, and in AccL for Exp = 7-14
;* (Right shift for Exp = 0-6, Left shift for Exp = 7-14)
;*
LAC    Temp2,9       ; Load exponent into high accumulator ...
SACH   Temp1         ; and store in Temp1
LACK   127           ; Load mask for lower 7 bits ...
AND    Temp2         ; and mask lower 7 bits (mantissa)
ADD    One,7         ; Add 1 (Q7) to get 1+x
SACL   Temp2         ; Store mantissa
LT     Temp2         ; Prepare to shift
LAC    Temp1         ; Read shift multiplier corresponding to ...
ADD    ShiftTableOffset ; the exponent from a table
TBLR   Temp3
MPY    Temp3         ; Multiply by shift constant
PAC
BLZ    LeftShift     ; Branch if left shift was required
SACH   Dq,1         ; Right shift 1 more, store in Dq (mag)
B      AddSign       ; Branch to add sign
LeftShift:

```

```

ABS          ; Convert to positive value
SACL   Dq    ; Save result in Dq (mag)
AddSign:
LAC    One,11
SACL   SDq   ; Save +1 (Q11) as Sgn(Dq)
LAC    I     ; Check sign of I (4th bit)
SUB    One,3
BLZ    HaveDq ; Branch if I +ve, SDq and Dq are correct
ZAC
SUB    Dq    ; Negate Dq, I is -ve
SACL   Dq    ; Store modified Dq
LAC    Minus,11
SACL   SDq   ; Save -1 (Q11) as Sgn(Dq)
HaveDq:
;*****
;QSFA:
;* Quantizer scale factor adaptation section. Using the previously obtained
;* quantized difference signal (I), compute the new scale factors (Yu, Y1).
;*****
;* First compute W[I] from the table
LAC    WITableOffset ; Load table offset
ADD    IM            ; Add magnitude of quantized difference
TBLR   Temp1         ; Lookup WI (Q4), store in Temp1
;*
;* Update the fast scale factor as:
;*  $Yu(k) = (1 - 2^{*-5}) * Y(k) + (2^{*-5}) * WI(k)$ 
;* Note: Yu and Y are (13b POS Q9), WI is (12b TC Q4 sign extended to 16b)
;*
LAC    Y,12         ; Load Y in Q21 form
SUB    Y,7          ; Subtract Y/32 (Q21)
ADD    Temp1,12     ; Add WI/32 (Q21)
SACH   Yu,4         ; Store Yu in (13b POS Q9) format
LAC    Yu           ; Reload Yu ...
AND    M8191        ; and mask lower 13 bits
SACL   Yu           ; Save final Yu
;*
;* Limit the fast scale factor  $1.06 \leq Yu \leq 10.0$ 
;* Q9 form ( $544 \leq Yu \leq 5120$ )
;*
SUB    K544         ; Check low threshold
BGEZ   CheckHi     ; Branch if above lower threshold
LAC    K544         ; Load lower threshold, as it was below
B      StoreYu      ; Branch to store lower limit
CheckHi:
SUB    K4576        ; acc - 5120, check upper threshold
BLEZ   GoodYu       ; Branch if within limits
LAC    K5120        ; Load upper threshold, as it was above
StoreYu:
SACL   Yu           ; Store the modified Yu
GoodYu:
;*
;* Update the slow scale factor as:
;*  $Y1(k) = (1 - 2^{*-6}) * Y1(k-1) + (2^{*-6}) * Yu(k)$ 
;* Note: Yu is (13b POS Q9), Y1 is (19b POS Q15)
;*
ZALH   Y1h          ; Load Y1 to accumulator
ADDS   Y1l
RPTK   5

```

```

SFL          Ylh          ; Shift Y1 left 6 bits (Q21)
SUBH        Ylh
SUBS        Y1l          ; Subtract (2**-6)*Y1 (Q21)
ADD         Yu,6          ; Add (2**-6)*Yu (Q21)
RPTK        5
SFR          ; Shift Y1 right 6 bits (Q15)
SACL        Y1l
SACH        Ylh          ; Save modified Y1
LACK        7            ; Mask 3 lower bits of Ylh
AND         Ylh
SACL        Ylh

;*****
SpeedControl:
;* Adaptation speed control section. Using the previously obtained quantized
;* signal magnitude (IM), compute the adaptation speed control (Ap).
;*****
;* First compute the F[I] function
;*
LAC          FITableOffset ; Look-up value of FI function
ADD          IM
TBLR        Temp1         ; Load FI value into Temp1 (Q4)

;* Now, update the short term average of FI as:
;*  $Dms(k) = (1 - 2**-5) * Dms(k-1) + (2**-5) * FI(k)$ 
;* Note: FI is (7b POS Q4), Dms is (12b POS Q9)
;*
LAC          Temp1,15      ; Load (2**-5)*FI (Q24)
ADD          Dms,15        ; Add Dms(k-1) (Q24)
SUB          Dms,10        ; Subtract (2**-5)*Dms (Q24)
SACH        Dms,1         ; Save the new Dms (Q9)

;* Update long term average of FI as:
;*  $Dml(k) = (1 - 2**-7) * Dml(k-1) + (2**-7) * FI(k)$ 
;* Note: FI is (7b POS Q4), Dml is (14b POS Q11)
;*
LAC          Temp1,15      ; Load (2**-7)*FI (Q26)
ADD          Dml,15        ; Add Dml(k-1) (Q26)
SUB          Dml,8         ; Subtract (2**-7)*Dml (Q26)
SACH        Dml,1         ; Save the new Dml (Q11)

;* Now compute the speed control parameter as:
;*  $Ap(k) = (1 - 2**-4) * Ap(k-1) + (2**-3) * \begin{cases} \text{if } (Y < 3) \text{ or } (|Dms - Dml| > (2**-3) * Dml) \\ (1 - 2**-4) * Ap(k-1) \\ \text{otherwise} \end{cases}$ 
;* Note: Dms is (12b POS Q9), Dml is (14b POS Q11), Ap is (10b POS Q8)
;*
ZALH        Ap            ; Load Ap (Q24)
SUB         Ap,12         ; Subtract (2**-4)*Ap (Q24)
SACH        Ap            ; Save (1 - 2**-4)*Ap (Q8)
LAC         Y
SUB         Three,9       ; Check if Y < 3 (Q9)
BLZ         AddEighth     ; Add 1/8 if so.
ZALH        Dms           ; Load Dms (Q25)
SUB         Dml,14        ; Subtract Dml (Q25)
ABS         ; Get magnitude of result
SUB         Dml,11        ; Check if greater than Dml/8 (Q25)
BLZ         GoodAp        ; Branch if not

```

```

AddEighth:
LAC          Ap            ; Reload Ap
ADD          One,5         ; Add 1/8 (Q8)
SACL        Ap            ; Save modified Ap
GoodAp:
;*
;*****
APred:
;* Adaptive predictor section. Using the previously obtained difference
;* signal (Dq), and the previous signal estimate (Se), compute the new
;* adaptive filter coefficients.
;*****
;* First, compute the reconstructed signal (Sr) as:
;*  $Sr(k) = Dq(k) + Se(k)$ 
;* Note: Dq, Se, and Sr are (16b TC Q0)
;*
LAC          Dq
ADD          Se
SACL        Sr            ; Save Sr = Dq + Se (Q0)

;* Compute coefficients of 6th order FIR predictor filter
;*  $Bi(k) = (1 - 2**-8) * Bi(k-1) + (2**-7) * \text{sgn}(Dq(k)) * \text{sgn}(Dq(k-1))$ 
;* for i = 1 .. 6 (Bi implicitly limited to +/- 2)
;* Note: Bi are (16b TC Q14), SDqi are (16b TC Q11; +1 = +2048, -1 = -2048)
;*
GetB6:      LT            SDq6
LAC          B6,15        ; Load old Bi (Q29)
SUB          B6,7         ; Subtract (2**-8)*Bi (Q29)
MPY          SDq          ; SDq * SDq6 * (2**-7) (Q29)
LTD          SDq5         ; Create new Bi, shift data for next sample
SACH        B6,1         ; Save new Bi in (Q14) format

GetB5:      LAC          B5,15
SUB          B5,7
MPY          SDq
LTD          SDq4
SACH        B5,1

GetB4:      LAC          B4,15
SUB          B4,7
MPY          SDq
LTD          SDq3
SACH        B4,1

GetB3:      LAC          B3,15
SUB          B3,7
MPY          SDq
LTD          SDq2
SACH        B3,1

GetB2:      LAC          B2,15
SUB          B2,7
MPY          SDq
LTD          SDq1
SACH        B2,1

GetB1:      LAC          B1,15
SUB          B1,7
MPY          SDq
LTD          SDq
SACH        B1,1

;*
;* Update coefficients of 2nd order IIR prediction filter
;*

```

```

;* First get sign of the sum of difference signal (Dq), and the partial signal
;* estimate (Sex).
;* Note: Dq and Sex are (16b TC Q0), Pk1 is (10b TC Q9 sign extended to 16b)
;*
DMOV Pk1 ; Shift Pk1-->Pk2
DMOV Pk0 ; Shift Pk0-->Pk1
LAC One,9
SACL Pk0 ; Store +1 (Q9) as initial sign
LAC Sex
ADD Dq ; Calculate (Sex + Dq)
BGEZ GoodPk ; Branch if result is +ve, already saved +1
LAC Minus,9
SACL Pk0 ; Result is -ve, save -1 (Q9)
GoodPk:
LT Pk0 ; Load Pk0 to T-reg in prep. for multiply
;*
;* Now calculate f[A1(k-1)] as:
;* f[A1(k-1)] = 4*A1, if |A1| <= 1/2
;* = 2*sgn(A1), if |A1| > 1/2
;* Note: A1 is (16b TC Q14), F is (16b TC Q13)
;*
LAC A1,1 ; Load 2*A1 (Q14) to accum
SACL Temp3 ; Save as initial F/2
BLZ NegA1 ; Branch if A1 is negative
SUB One,14 ; Calculate 2*(A1 - 1/2) (Q14)
BLZ HaveF ; F/2 is ok if |A1| < 1/2
LAC One,14 ; |A1| > 1/2, so load limit 1 (Q14)
B SaveF
NegA1:
ABS ; Make A1 positive
SUB One,14 ; Check if |A1| < 1/2
BLZ HaveF ; F/2 is ok if so
LAC Minus,14 ; |A1| > 1/2, so load limit -1 (Q14)
SaveF:
SACL Temp3 ; Save modified F/2
HaveF:
;*
;* Now calculate A1(k) as:
;* A1(k) = (1 - 2**(-8))*A1(k-1) + 3*(2**(-8))*sgn[P(k)]*sgn[P(k-1)]
;* Note: A1 is (16b TC Q14), sgn(p) is (10b TC Q9 sign extended to 16b)
;*
LAC A1,12 ; Load A1 (Q26)
SUB A1,4 ; Subtract (2**(-8))*A1 (Q26)
MPY Pk1 ; Calculate sgn(Pk1)*sgn(Pk), sgn(Pk) is ...
APAC ; in T register already. Result is (Q18)
APAC ; Add 3*(2**(-8))*sgn(Pk1)*sgn(Pk) (Q26)
SACH A1,4 ; Save new A1 in (Q14)
PAC ; Acc has sgn[p(k-1)]*sgn[p(k)]
;*
;* Now calculate A2(k) as:
;* A2(k) = (1 - 2**(-7))*A2(k-1) +
;* + (2**(-7))*sgn[P(k)]*sgn[P(k-2)] - f[A1(k-1)]*sgn[P(k)]*sgn[P(k-1)]
;* Note: A2 is (16b TC Q14), sgn(p) is (10b TC Q9), F is (16b TC Q13)
;*
BGEZ SubtractF ; If sgn(Pk1)*sgn(Pk) +ve then subtract F
ZAC ; Otherwise, negate F then subtract
SUB Temp3 ; Temp3 has F/2
SACL Temp3 ; Save negated F/2
SubtractF:

```

```

LAC A2,12 ; Load old A2 (Q26)
SUB A2,5 ; Subtract (2**(-7))*A2 (Q26)
MPY Pk2 ; Calculate sgn(Pk)*sgn(Pk2) (Q18)
APAC
APAC ; Add (2**(-7))*sgn(Pk)*sgn(Pk2) (Q26)
SUB Temp3,6 ; Subtract (2**(-7))*F (Q26)
SACH A2,4 ; Save new A2 (Q14)
;*
;* Now limit A2 to +/- 0.75, and prevent overflow.
;* Note: A2 is (16b TC Q14)
;*
LAC A2
ABS ; Take absolute value of A2
SUB Thre,12 ; Check that |A2| < 0.75 (Q14)
BLEZ GoodA2 ; Branch if A2 is within limits
LAC A2
BGEZ A2PosLimit ; Save +ve limit if A2 is +ve
A2NegLimit:
LAC Minus3,12 ; Save -ve limit of -0.75 (Q14)
B SaveA2
A2PosLimit:
LAC Thre,12 ; Save +ve limit of +0.75 (Q14)
SaveA2:
SACL A2 ; Save limited A2
GoodA2:
;*
;* Now, limit A1 to +/- (1 - 2**(-4) - A2)
;* Note: A1 and A2 are (16b TC Q14)
;*
LAC M15,10 ; Load (1 - 2**(-4)) (Q14)
SUB A2 ; Subtract A2 (Q14)
SACL Temp1 ; Save limit of (1 - 2**(-4) - A2) in Temp1
LAC A1
ABS ; Load |A1|
SUB Temp1 ; Compare with limit
BLEZ GoodA1 ; Branch if |A1| <= limit
LAC A1
BGEZ LimitA1 ; Save positive limit if A1 > 0
ZAC
SUB Temp1 ; Calculate negative limit
SACL Temp1 ; Save negative limit
LimitA1:
LAC Temp1 ; Load limit value to accumulator
SACL A1 ; Store limited A1 value
GoodA1:
;*
;*****
;* The decoding is complete. Save the reconstructed signal, and return to
;* the calling routine.
;*
LAC Sr
SACL Sample ; Save the reconstructed signal
LAC Sample
LDPK 0 ; Switch back to page 0 variables
SACL LinVal ; Save the linear value
RET ; Return to calling routine
;*
;*****
InitAdpcmExp:

```

```

;* Routine to initialize the constants used by the Adpcm algorithm, and
;* reset the Adpcm algorithm variables. This routine must be called before
;* the first Adpcm conversion.
;*
;* Assumes:
;* DP = 0      DP is set to equal page 0 upon exit
;* ?SXM       Sign extension can be either on or off
;*****
;*
;*      LDPK      AdpcmPage      ; Switch to Adpcm variables
;*
;* Define basic constants
;*
;*      LACK      1
;*      SACL      One
;*      SUB       One,1
;*      SACL      Minus
;*      LACK      3
;*      SACL      Three
;*      SUB       Three,1
;*      SACL      Minus3
;*
;* Define bit masks
;*
;*      LACK      15
;*      SACL      M15
;*      LACK      127
;*      SACL      M127
;*      LAC       One,10
;*      SUB       One
;*      SACL      M1023
;*      ADD       One,10
;*      SACL      M2047
;*      ADD       One,11
;*      SACL      M4095
;*      ADD       One,12
;*      SACL      M8191
;*
;* Define ADPCM specific constants
;*
;*      LALK      544
;*      SACL      K544
;*      LALK      5120
;*      SACL      K5120
;*      LALK      4576
;*      SACL      K4576
;*      LALK      2309
;*      SACL      K2309
;*      LACK      138
;*      SACL      K138
;*      LACK      130
;*      SACL      K130
;*      LACK      79
;*      SACL      K79
;*      LACK      95
;*      SACL      K95
;*      LACK      46
;*      SACL      K46
;*      LACK      49
;*      SACL      K49
;*
```

```

;* Initialize data table pointers
;*
;*      LT       One
;*      MPYK      InvQTable
;*      PAC
;*      SACL      InvQTableOffset
;*      MPYK      ShiftTable
;*      PAC
;*      SACL      ShiftTableOffset
;*      MPYK      WITable
;*      PAC
;*      SACL      WITableOffset
;*      MPYK      FITable
;*      PAC
;*      SACL      FITableOffset
;*
;* Fall through to ResetAdpcmExp to reset the ADPCM variables
;*
;*****
ResetAdpcmExp:
;* Routine to reset the ADPCM compression algorithm variables.
;*
;* Assumes:
;* DP = 0      DP is set to page 0 upon exit
;* ?SXM       Sign extension can be either on or off
;*****
;*
;*      LDPK      AdpcmPage      ; Switch to Adpcm variables
;*
;* Zero the predictor coefficients, and delayed values
;*
;*      ZAC
;*      SACL      Dq
;*      SACL      Dq1
;*      SACL      Dq2
;*      SACL      Dq3
;*      SACL      Dq4
;*      SACL      Dq5
;*      SACL      B1
;*      SACL      B2
;*      SACL      B3
;*      SACL      B4
;*      SACL      B5
;*      SACL      B6
;*      SACL      Ap
;*      SACL      Sr
;*      SACL      Srl
;*      SACL      A1
;*      SACL      A2
;*      SACL      Dms
;*      SACL      Dml
;*      SACL      Ylh
;*
;* Set default values for the scale factors
;*
;*      LAC       K544
;*      SACL      Yu
;*      LAC       One,15
;*      ADD       One,11
;*      SACL      Y11
;*
```



```

;*
;* Initialize the signs of delayed values
;*
LAC      One,11
SACL    SDq1
SACL    SDq2
SACL    SDq3
SACL    SDq4
SACL    SDq5
SACL    SDq6

;*
LAC      One,9
SACL    Pk0
SACL    Pk1

;*
LDPK      0          ; Switch back to page 0 variables
RET

;*****
.data
;*****
;* Initialized data tables
;*
;* Inverse quantization table
InvQTable: ; (16b TC Q7)
.word 65401 ; |I| = 0 (-1.05)
.word 68 ; |I| = 1 (0.53)
.word 165 ; |I| = 2 (1.29)
.word 232 ; |I| = 3 (1.81)
.word 285 ; |I| = 4 (2.23)
.word 332 ; |I| = 5 (2.59)
.word 377 ; |I| = 6 (2.95)
.word 428 ; |I| = 7 (3.34)

;*
;* Log to linear shift factor table
ShiftTable: ; (16b TC Q0)
.word 256 ; Exp = 0 (Shift 9) - 1
.word 512 ; Exp = 1 (Shift 10) - 1
.word 1024 ; Exp = 2 (Shift 11) - 1
.word 2048 ; Exp = 3 (Shift 12) - 1
.word 4096 ; Exp = 4 (Shift 13) - 1
.word 8192 ; Exp = 5 (Shift 14) - 1
.word 16384 ; Exp = 6 (Shift 15) - 1
.word -1 ; Exp = 7 (Shift 0)
.word -2 ; Exp = 8 (Shift 1)
.word -4 ; Exp = 9 (Shift 2)
.word -8 ; Exp = 10 (Shift 3)
.word -16 ; Exp = 11 (Shift 4)
.word -32 ; Exp = 12 (Shift 5)
.word -64 ; Exp = 13 (Shift 6)
.word -128 ; Exp = 14 (Shift 7)

;*
;* Scale factor multiplier table
WITable: ; (16b TC Q4)
.word 65524 ; |I| = 0 (-0.75)
.word 4 ; |I| = 1 (0.25)
.word 27 ; |I| = 2 (1.69)
.word 50 ; |I| = 3 (3.12)

.word 98 ; |I| = 4 (6.12)
.word 184 ; |I| = 5 (11.50)
.word 340 ; |I| = 6 (21.25)
.word 1108 ; |I| = 7 (69.25)

;*
;* Rate of change weighting function table
FITable: ; (16b POS Q4)
.word 0 ; |I| = 0 (0)
.word 0 ; |I| = 1 (0)
.word 0 ; |I| = 2 (0)
.word 16 ; |I| = 3 (1)
.word 16 ; |I| = 4 (1)
.word 16 ; |I| = 5 (1)
.word 48 ; |I| = 6 (3)
.word 112 ; |I| = 7 (7)

;*****
;*
;* Variable declarations
;*
.bss A1,1 ; Coefficients for 2nd order IIR predictor
.bss A2,1 ; ...
.bss B1,1 ; Coefficients for 6th order FIR predictor
.bss B2,1 ; ...
.bss B3,1 ; ...
.bss B4,1 ; ...
.bss B5,1 ; ...
.bss B6,1 ; ...
.bss Dq,1 ; Inverse quantized difference signal
.bss Dq1,1 ; Delayed versions of Dq
.bss Dq2,1 ; ...
.bss Dq3,1 ; ...
.bss Dq4,1 ; ...
.bss Dq5,1 ; ...
.bss SDq,1 ; Sign of inverse quantized difference signal
.bss SDq1,1 ; Delayed versions of SDq
.bss SDq2,1 ; ...
.bss SDq3,1 ; ...
.bss SDq4,1 ; ...
.bss SDq5,1 ; ...
.bss SDq6,1 ; ...
.bss Sr,1 ; Reconstructed signal estimate
.bss Sr1,1 ; Delayed version of Sr
.bss Pk0,1 ; Sign of partial reconstructed signal
.bss Pk1,1 ; Delayed versions of Pk0
.bss Pk2,1 ; ...
.bss Se,1 ; Signal estimate
.bss Sez,1 ; Partial signal estimate
.bss Y1l,1 ; Low word of unlocked scale factor
.bss Y1h,1 ; High word of unlocked scale factor
.bss Yu,1 ; Locked scale factor
.bss Y,1 ; Adaptive quantizer scale factor
.bss Dms,1 ; Short term average of ADPCM signal
.bss Dml,1 ; Long term average of ADPCM signal
.bss Ap,1 ; Unlimited speed control factor
.bss Al,1 ; Limited speed control factor
.bss I,1 ; Quantized difference signal
.bss IM,1 ; Quantized difference signal magnitude
.bss D,1 ; Difference signal
.bss Temp1,1 ; Temporary scratch variables

```

```

.bss Temp2,1      ; ...
.bss Temp3,1      ; ...
.bss Sample,1     ; Input linear speech sample
;*
;* Reserve space for constants
;*
.bss InvQTableOffset,1 ; Offset to Inverse Quantization table
.bss ShiftTableOffset,1 ; Offset to Log-to-Linear shift table
.bss WTableOffset,1   ; Offset to Scale Factor Multiplier table
.bss FTableOffset,1   ; Offset to Rate-of-Change Weighting table
.bss One,1            ; 1
.bss Three,1          ; 3
.bss Minus,1          ; -1
.bss Minus3,1         ; -3
.bss M15,1            ; 15 (000F) 4 bit mask
.bss M127,1           ; 127 (007F) 7 bit mask
.bss M1023,1          ; 1023 (03FF) 10 bit mask
.bss M2047,1          ; 2047 (07FF) 11 bit mask
.bss M4095,1          ; 4095 (0FFF) 12 bit mask
.bss M8191,1          ; 8191 (3FFF) 13 bit mask
.bss K2309,1          ; 2309
.bss K138,1           ; 138
.bss K130,1           ; 130
.bss K79,1            ; 79
.bss K95,1            ; 95
.bss K46,1            ; 46
.bss K49,1            ; 49
.bss K544,1           ; 544
.bss K4576,1          ; 4576
.bss K5120,1          ; 5120
;*
*****
.end
*****

```

B.2.2 PC Host Program Listings

The following pages contain the C language program listings for the IBM-PC/AT host computer. These programs are used to transfer compressed speech data between the PC's hard-disk, and the data memory of the TMS320 processor system. There are separate programs for speech storage and speech playback; this was done for simplicity, and it would be a straight-forward task to combine the two separate programs into one. Also included are the C language program listings used to download TMS320 software to the TMS320c25 processor system.

This file contains the programs that implement the transfer of data from (SilRec) and to (SilPlay) the TMS320c25 processor system. These programs assume that the appropriate TMS320 program has already been downloaded to the TMS320c25 processor system.

TmsProc.h

```

/*
** TMSPROC.H
** November 30, 1989
** Chris Rose
**
** Include file defining constants for use with the TMS320c25 processor
** board.
*/

/*
** Define memory locations for accessing the TMS320 processor board.
*/
#define TMSegment      0xD000 /* Base address for board 0xD0000 */
#define ControlWrite   0xE000 /* Offset for control register */
#define StatusRead     0xE000 /* Offset for status register */
#define ProgramMem     0x0000 /* Offset for TMS external program memory */
#define TMSDataOffset  0x8000 /* Offset for TMS external data memory */
#define DataSize       0x2000 /* Size in bytes of external data mem */
#define TMSCommLatch   0xF000 /* Offset for communication latch word */
#define LowCommLatch   0xF000 /* Offset for comm. latch low byte */
#define HighCommLatch  0xF001 /* Offset for comm. latch high byte */
/*
** Defines for use with the ControlWrite register. The bits are set up
** as follows:
**
** b0 - TMS hold request (active low)
** b1 - TMS software reset (active low)
** b2 - TMS Bio input line
** b3 - Interrupt 0 (active low)
** b4 - Interrupt 1 (active low)
** b5 - Interrupt 2 (active low)
** b6 - unused
** b7 - unused
**
** WrBase describes the 'default' state for the control register while the
** processor is running. The actual value to write to the control register
** is obtained by ANDing the WrBase value with any other action to be taken,
** such as WrHold to put the TMS on hold (ie, WrBase & WrHold).
*/
#define WrBase         0x3F
#define WrHold         0xFE
#define WrReset        0xFD
#define WrBioLow       0xFB
#define WrInt0         0xF7
#define WrInt1         0xEF
#define WrInt2         0xDF
/*
** Defines for the TMS status register. This register is set-up as follows:
**
** b0 - TMS hold acknowledge (active low)
** b1 - TMS External Flag
** b2 - Reset signal applied to TMS (active low)

```

```

**
** b3 - Bio signal applied to TMS
** b4 - Interrupt 0 applied to TMS (active low)
** b5 - Interrupt 1 applied to TMS (active low)
** b6 - Interrupt 2 applied to TMS (active low)
** b7 - unused
*/
#define RdHoldAck      0x01
#define RdXF           0x02
#define RdReset        0x04
#define RdBio         0x08
#define RdInt0         0x10
#define RdInt1         0x20
#define RdInt2         0x40

```

SilRec.c

```

/*
** SilRec.c
** February 12, 1990
** Chris Rose
**
** PC disk interface program for the silence deletion algorithm.
*/

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>

#include "include\tmsproc.h"

#define WRITEFILEMODE  O_WRONLY|O_BINARY|O_CREAT|O_TRUNC|S_IWRITE
#define TMSINDEX        (peekb(TMSegment, TMSCommLatch))
#define MAXBLOCKS       2500
#define BLOCKSIZE       2048

int WriteBlock( unsigned FileHandle, unsigned DataOffset);

/*****
void main( int Argc, char *Argv[] )
*****/
{
    unsigned Index, NumBlocks, DataPointer;
    int FileHandle;
    char Response;

    if (Argc < 2) {
        printf("Usage: SIL5REC filename [/]\n");
        exit(1);
    }

    if ((FileHandle = open(Argv[1], WRITEFILEMODE)) < 0) {
        printf("Error: Cannot open file \"%s\"\n", Argv[1]);
        exit(1);
    }
}

```

```

printf("Ready to begin recording. Continue [Y/N] ...");
Response = getche();
if ((Response != 'y') && (Response != 'Y')) {
    close(FileHandle);
    exit(1);
}
printf("\nRecording ... \n");

/* Reset the TMS320 */
pokeb(TMSSegment, ControlWrite, (WrBase & WrReset & WrBioLow));

NumBlocks = 1;
Index = 1;
DataPointer = TMSDataOffset;

/* Start the TMS running */
pokeb(TMSSegment, ControlWrite, (WrBase & WrBioLow));

while ((!kbhit()) && (NumBlocks < MAXBLOCKS)) {
    if (Index == TMSINDEX) {
        NumBlocks++;
        if (WriteBlock(FileHandle, DataPointer) < 0) {
            printf("Error: Cannot write data file\n");
            pokeb(TMSSegment, ControlWrite, (WrBase & WrHold));
            exit(1);
        }
        if (Index == 0) {
            DataPointer = TMSDataOffset;
        }
        else {
            DataPointer += BLOCKSIZE;
        }
        Index++;
        Index %= 0x03;
    }
}
if (kbhit()) getch();

/* Signal to TMS to stop sampling and wait for XF */
pokeb(TMSSegment, ControlWrite, (WrBase)); /* BIO is high */
while ((peekb(TMSSegment, StatusRead) & RdXF) == 0) {
    printf("Waiting for TMS to stop ... \n");
}
pokeb(TMSSegment, ControlWrite, (WrBase & WrHold));
printf("TMS stopped\n");

/* Write any outstanding data to disk */
if (WriteBlock(FileHandle, DataPointer) < 0) {
    printf("Error: Cannot write data file\n");
    pokeb(TMSSegment, ControlWrite, (WrBase & WrHold));
    exit(1);
}
if (Index == TMSINDEX) {
    if (Index == 0) {
        DataPointer = TMSDataOffset;
    }
    else {
        DataPointer += BLOCKSIZE;
    }
}

```

```

    if (WriteBlock(FileHandle, DataPointer) < 0) {
        printf("Error: Cannot write data file\n");
        pokeb(TMSSegment, ControlWrite, (WrBase & WrHold));
        exit(1);
    }
}

close(FileHandle);
exit(0);
}

/*
** WriteBlock
** Routine to transfer data from a memory location to disk. It uses
** a low level DOS routine for speed.
*/
/*****
int WriteBlock( unsigned FileHandle, unsigned DataOffset)
*****/
{
    union REGS Regs;
    struct SREGS SRegs;
    int Status;

    /* Use DOS interrupt 0x40 to write the block of data to the disk */
    Regs.h.ah = 0x40;
    Regs.x.bx = FileHandle;
    Regs.x.cx = BLOCKSIZE; /* Number of bytes to write */
    Regs.x.dx = DataOffset; /* Offset to start of data to transfer */
    SRegs.ds = TMSSegment; /* Segment of data */

    if ((Status = intdosx(&Regs, &Regs, &SRegs)) == 0) {
        Status = -1; /* Error code for disk full */
        return(Status);
    }

    /* If carry flag set, return the error code (AX) from intdosx */
    return(Regs.x.cflag ? Status : 0);
}

```

SilPlay.c

```

/*
** SilPlay.c
** February 12, 1990
** Chris Rose
**
** PC disk interface program for the silence deletion playback algorithm.
*/

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>

```

```

#include "include\tmsproc.h"

#define READFILEMODE    O_RDONLY|O_BINARY
#define TMSINDEX        (peek(TMSegment, TMSCommLatch))
#define BLOCKSIZE       2048

int ReadBlock( unsigned FileHandle, unsigned DataOffset );

/*****
void main( int Argc, char *Argv[] )
/*****/
{
    unsigned Index, DataPointer;
    int FileHandle, Status, NumTimes;
    char Response;

    if (Argc < 2) {
        printf("Usage: SIL5PLAY filename [/\n");
        exit(1);
    }

    if ((FileHandle = open(Argv[1], READFILEMODE)) < 0) {
        printf("Error: Cannot open file \"%s\"\\n", Argv[1]);
        exit(1);
    }

    /* Fill TMS data memory first */
    DataPointer = TMSDataOffset;
    NumTimes = 0;
    do {
        Status = ReadBlock(FileHandle, DataPointer);
        DataPointer += BLOCKSIZE;
        NumTimes++;
    } while ((Status >= 0) && (NumTimes < 4));

    printf("Ready to begin playback. Continue [Y/N] ...");
    Response = getche();
    if ((Response != 'y') && (Response != 'Y')) {
        close(FileHandle);
        exit(1);
    }
    printf("\\nPlaying ...\\n");

    /* Reset the TMS320 */
    pokeb(TMSegment, ControlWrite, (WrBase & WrReset));

    Index = 1;
    DataPointer = TMSDataOffset;

    /* Start the TMS320 running */
    pokeb(TMSegment, ControlWrite, (WrBase));

    while ((!kbhit()) && (Status >= 0)) {
        if (Index == TMSINDEX) {
            Status = ReadBlock(FileHandle, DataPointer);
            if (Index == 0) {
                DataPointer = TMSDataOffset;
            }
            else {
                DataPointer += BLOCKSIZE;
            }
        }
    }
}

```

```

        }
        Index++;
        Index &= 0x03;
    }
}

if (kbhit()) {
    getch();
    pokeb(TMSegment, ControlWrite, (WrBase & WrHold));
    printf("TMS forced to stop\\n");
}
else {
    /* Wait for TMS to finish playing file */
    while ((peekb(TMSegment, StatusRead) & RdXF) == 0) {
        printf("Waiting for TMS to finish ...\\n");
    }
    pokeb(TMSegment, ControlWrite, (WrBase & WrHold));
    printf("TMS stopped\\n");
}

close(FileHandle);
exit(0);
}

/*
** ReadBlock
** Routine to transfer data from disk to a memory location. It uses
** a low level DOS routine for speed.
*/
/*****/
int ReadBlock( unsigned FileHandle, unsigned DataOffset )
/*****/
{
    union REGS Regs;
    struct SREGS SRegs;
    int Status;

    /* Use DOS interrupt 0x3F to read the block of data from the disk */
    Regs.h.ah = 0x3F;
    Regs.x.bx = FileHandle;
    Regs.x.cx = BLOCKSIZE; /* Number of bytes to read */
    Regs.x.dx = DataOffset; /* Offset to start of data to transfer */
    SRegs.ds = TMSegment; /* Segment of data */

    if ((Status = intdosx(&Regs, &Regs, &SRegs)) == 0) {
        Status = -1; /* Error code for end of file */
        return(Status);
    }

    /* If carry flag set, return the error code (AX) from intdosx */
    return(Regs.x.cflag ? Status : 0);
}

```

This file contains the program used to download programs to the TMS320C25 processor system.

TmsLoad.c

```

/*
** TMSLOAD.C
** November 24, 1989
** Chris Rose
**
** This program is used to load a TI COFF format object file onto the
** TMS320C25 processor board. It uses the TI utility DSPROM to first
** convert the COFF file into two Intel Hex format files (one for the
** MSByte, one for the LSByte), which are then loaded into the processor
** boards program memory.
**
** The Intel Hex format files have lines with the following format:
**      :NNAAAAXX<----Data---->CC
** where:
** NN is an unsigned integer that contains the number of data bytes on
** the line.
** AAAA is an unsigned integer (high byte first) that contains the start
** address that the following data should be loaded at.
** XX is a code that describes the contents of the line.
** CC is a check sum for the line (byte adding line + check sum should give
** zero with a carry.
**
** NOTE: all information in the files is in HEX format, and can be read with
** "scanf("%2x", &ByteValue).
*/

#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <string.h>

#define True      1
#define False     0
#define MAXNAMELEN 8
#define MAXLINELEN 100
#define TMSSEGMENT 0xD000
#define PROGOFFSET 0x0000
#define CTRLADDR 0xE000
#define RESETHOLD 0x3C
#define HOLDA 0x01

int CorrectFileName(char *FileName, char *CorrectedName);
int ConvertFile(char *FileName);
int LoadFiles(char *FileName);
int ReadLine(unsigned int *BinaryLine, unsigned int *Address, FILE *HiFile,
FILE *LoFile);
unsigned int Read2Bytes(FILE *LoFile, FILE *HiFile);
unsigned int ReadByte(FILE *InFile);

/*****
int main(argc, argv)
int argc;
char *argv[];

```

```

*****/
{
    int Status;
    char FileName[MAXNAMELEN+5];

    if (argc < 2) {
        fprintf(stderr, "Error: Insufficient arguments.\n");
        fprintf(stderr, "Usage: TMSLOAD <TMS executable object file>\n");
        fprintf(stderr, "        Downloads the named file to the TMS320");
        fprintf(stderr, " processor board.\n");
        exit(EXIT_FAILURE);
    }

    Status = CorrectFileName(argv[1], FileName);
    if (Status) {
        fprintf(stderr, "Error: Incorrect input File Name format.\n");
        fprintf(stderr, "FileNames should have a maximum of 8 letters, with");
        fprintf(stderr, " no filename\n extension (.OUT is added)\n");
        exit(EXIT_FAILURE);
    }

    fprintf(stderr, "\nConverting COFF file '%s.OUT'", FileName);
    fprintf(stderr, " to Intel Hex format ...\n\n");
    Status = ConvertFile(FileName);
    if (Status) {
        fprintf(stderr, "\nError: Could not convert input file");
        fprintf(stderr, " '%s'\n", strcat(FileName, ".OUT"));
        exit(EXIT_FAILURE);
    }
    fprintf(stderr, "\nSuccessful conversion completed.\n");

    fprintf(stderr, "\nLoading files onto TMS processor board ...\n");
    Status = LoadFiles(FileName);
    if (Status) {
        fprintf(stderr, "\nError: Could not load file to TMS board\n");
        exit(EXIT_FAILURE);
    }
    fprintf(stderr, "\nSuccessful completion of Loading.\n");

    exit(EXIT_SUCCESS);
}

/*
** CorrectFileName
**
** This routine will check the passed filename for format, while copying
** it to a new variable location.
**
*****/
int CorrectFileName(PassedName, NewName)
char PassedName[];
char NewName[];
/*****
{
    int Index;

    Index = 0;
    while ((Index < MAXNAMELEN) && (PassedName[Index] != '.') &&
(PassedName[Index])) {
        NewName[Index] = PassedName[Index];

```

```

        Index++;
    }
    NewName[Index] = '\0';
    strcpy(NewName);

    if ((PassedName[Index] == '.') || (PassedName[Index] == '\0')) {
        return(0);
    }
    else {
        return(1);
    }
}

/*
** ConvertFile
**
** This routine will execute the TI utility DSPROM to convert the file with
** the passed FileName (in TI-COFF object format) to two Intel Hex format
** files with the names FileName.HI and FileName.LO.
*/
/*****
int ConvertFile(FileName)
char *FileName;
*****/
{
    int Status;
    char ObjFile[MAXNAMELEN+5];

    strcpy(ObjFile, FileName);
    strcat(ObjFile, ".OUT");
    Status = spawnlp(P_WAIT, "dsprom.exe", "dsprom.exe", "-i", ObjFile, NULL);
    return(Status);
}

/*
** LoadFiles
**
** This routine will take the two Intel Hex format files FileName.HI and
** FileName.LO and download them to the TMS processor board.
*/
/*****
int LoadFiles(FileName)
char *FileName;
*****/
{
    char HiFileName[MAXNAMELEN+5], LoFileName[MAXNAMELEN+5];
    char Done, ErrorStatus;
    unsigned int Address, TempAddress, BinaryLine[MAXLINELEN], Test;
    int NumWords, Index;
    FILE *HiFile, *LoFile;

    strcpy(HiFileName, FileName);
    strcpy(LoFileName, FileName);
    strcat(HiFileName, ".HI");
    strcat(LoFileName, ".LO");

    if ((HiFile = fopen(HiFileName, "rt")) == NULL) {
        fprintf(stderr, "\nError: Cannot open file '%s' ", HiFileName);
        fprintf(stderr, "for reading.\n");
        return(1);
    }

```

```

    }

    if ((LoFile = fopen(LoFileName, "rt")) == NULL) {
        fprintf(stderr, "\nError: Cannot open file '%s' ", LoFileName);
        fprintf(stderr, "for reading.\n");
        fclose(HiFile);
        return(1);
    }

    /* Put TMS320 Processor on Hold and wait for acknowledgement */
    pokeb(TMSSEGMENT, CTRLADDR, RESETOLD);
    while (peekb(TMSSEGMENT, CTRLADDR) & HOLDA);

    ErrorStatus = 0;
    Done = 0;
    while (!Done) {
        NumWords = ReadLine(BinaryLine, &Address, HiFile, LoFile);
        if (NumWords <= 0) {
            Done = 1;
            if (NumWords < 0) {
                ErrorStatus = 1;
            }
        }
        else {
            /* Write the data to the TMS board */
            TempAddress = Address;
            for (Index = 0; Index < NumWords; Index++) {
                pokeb(TMSSEGMENT, (PROGOFFSET+TempAddress), BinaryLine[Index]);
                TempAddress += 2;
            }
            /* Check that data is correct */
            TempAddress = Address;
            for (Index = 0; Index < NumWords; Index++) {
                Test = peekb(TMSSEGMENT, (PROGOFFSET+TempAddress));
                TempAddress += 2;
                if (Test != BinaryLine[Index]) {
                    ErrorStatus = 1;
                    Done = 1;
                }
            }
        }
    }

    fclose(HiFile);
    fclose(LoFile);
    system("del *.hi");
    system("del *.lo");

    return(ErrorStatus);
}

/*
** ReadLine
**
** This routine will create an array of 16 bit words of binary data created
** by reading a line of Hex data from each of the two Intel Hex format files.
*/
/*****
int ReadLine(BinaryLine, StoreAddress, HiFile, LoFile)
unsigned int BinaryLine[];
*****/

```



```

unsigned int *StoreAddress;
FILE *HiFile;
FILE *LoFile;
/*****
{
    char ReadChar, Error;
    unsigned int Address, NumToRead;
    int Index;

    Error = False;
    /* Move the file pointers to the start of the next line */
    do {
        ReadChar = fgetc(HiFile);
        if (ReadChar == EOF) {
            Error = True;
        }
    } while ((ReadChar != ':') && (Error == False));
    do {
        ReadChar = fgetc(LoFile);
        if (ReadChar == EOF) {
            Error = True;
        }
    } while ((ReadChar != ':') && (Error == False));
    if (Error) {
        return(-1);
    }

    /* Read in the number of bytes per line */
    NumToRead = ReadByte(HiFile);
    if (NumToRead != ReadByte(LoFile)) {
        return(-1);
    }

    /* Read in the address at which to data is to be stored */
    Address = Read2Bytes(HiFile, HiFile);
    if (Address != Read2Bytes(LoFile, LoFile)) {
        return(-1);
    }

    /* Skip over unused code bytes */
    ReadByte(HiFile);
    ReadByte(LoFile);

    /* Write address for loading data to, converting from 8bit data to
       16 bit data by multiplying address by 2 */
    *StoreAddress = (Address << 1);

    /* Read the data from the files */
    for (Index = 0; Index < NumToRead; Index++) {
        BinaryLine[Index] = Read2Bytes(LoFile, HiFile);
    }

    return((unsigned int) NumToRead);
}

/*
** Read2Bytes
**
** This routine will read a byte value from each of 2 files, and will form
** a 16 bit word (unsigned int) with the result. Both bytes can be read from

```

```

** the same file by providing the same Stream identifier as both arguments.
** The routine reads the High order byte first.
*/
/*****
unsigned int Read2Bytes(LowFile, HighFile)
    FILE *LowFile;      /* File to read low byte from */
    FILE *HighFile;     /* File to read high byte from */
/*****
{
    unsigned int LowByte, HighByte, Number;

    fscanf(HighFile, "%2x", &HighByte);
    fscanf(LowFile, "%2x", &LowByte);
    Number = (HighByte << 8) | (LowByte);
    return(Number);
}

/*****
unsigned int ReadByte(InFile)
    FILE *InFile;
/*****
{
    unsigned int Number;

    fscanf(InFile, "%2x", &Number);
    return(Number);
}

```